2012-03

Computing the Algebraic Immunity of Boolean Functions on the SRC-6 Reconfigurable Computer

McCay, Matthew Eric

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

COMPUTING THE ALGEBRAIC IMMUNITY OF BOOLEAN FUNCTIONS ON THE SRC-6 RECONFIGURABLE COMPUTER

by

Matthew Eric McCay

March 2012

Thesis Co-Advisors: Jon T. Butler
Pantelimon Stanica

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2012	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Computing the Algebraic Immunity of Boolean Functions on the SRC-6 Reconfigurable Computer	5. FUNDING NUMBERS
6. AUTHOR(S) Matthew Eric McCay	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number _____N/A_____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

Boolean functions with high algebraic immunity (AI) are vital in reducing the possibility of utilizing algebraic attacks to break an encryption system. Simple algorithms exist to compute the AI of a given *n*-variable Boolean function, but the time required to test a large number of functions is much greater on conventional computing systems. AI was computed for all functions through $n = 5$ using the SRC-6. AI was also computed for $n = 5$ using a C algorithm. The SRC-6 performed 4.86 times faster than a conventional processor for this computation. It is believed that this is the first enumeration of all 5-variable functions with respect to AI.

Monte Carlo trials were performed for $n = 6$, both on the SRC-6 and utilizing a C algorithm on a conventional processor. These trials provided the first known distribution of AI for 6-variable functions.

Some algorithms for computing AI require a conversion between the truth table form of the function and its algebraic normal form. The first known Verilog implementation of a reduced transeunt triangle was developed for this conversion. This reduced form requires many fewer gates and has $O(n)$ delay versus $O(2^n)$ delay for a full transeunt triangle.

14. SUBJECT TERMS Algebraic Immunity, Cryptography, Boolean Functions, Transeunt Triangle, Reconfigurable Computing, SRC-6, FPGA, Verilog, Algebraic Attack	15. NUMBER OF PAGES 172
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

COMPUTING THE ALGEBRAIC IMMUNITY OF BOOLEAN FUNCTIONS ON THE SRC-6 RECONFIGURABLE COMPUTER

Matthew Eric McCay
Lieutenant, United States Navy
B.S., University of Illinois at Urbana-Champaign, 2005

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

and

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
March 2012

Author: Matthew Eric McCay

Approved by: Dr. Jon T. Butler
Thesis Co-Advisor

Dr. Pantelimon Stanica
Thesis Co-Advisor

Dr. Clark Robertson
Chair, Department of Electrical and Computer Engineering

Dr. Carlos Borges
Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Boolean functions with high algebraic immunity (AI) are vital in reducing the possibility of utilizing algebraic attacks to break an encryption system. Simple algorithms exist to compute the AI of a given n-variable Boolean function, but the time required to test a large number of functions is much greater on conventional computing systems. AI was computed for all functions through $n = 5$ using the SRC-6. AI was also computed for $n = 5$ using a C algorithm. The SRC-6 performed 4.86 times faster than a conventional processor for this computation. It is believed that this is the first enumeration of all 5-variable functions with respect to AI.

Monte Carlo trials were performed for $n = 6$, both on the SRC-6 and utilizing a C algorithm on a conventional processor. These trials provided the first known distribution of AI for 6-variable functions.

Some algorithms for computing AI require a conversion between the truth table form of the function and its algebraic normal form. The first known Verilog implementation of a reduced transeunt triangle was developed for this conversion. This reduced form requires many fewer gates and has $O(n)$ delay versus $O(2^n)$ delay for a full transeunt triangle.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AI	Algebraic Immunity
ANF	Algebraic Normal Form
FPGA	Field Programmable Gate Array
FUT	Function Under Test
TT	Truth Table
XOR	Exclusive Or

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Computer security is a topic of extreme importance in the Information Age. The basis for secure communications is provably secure cryptographic systems. There are many cryptographic properties that characterize the security of a particular system. One of the more recent and high profile properties is algebraic immunity (AI).

To understand algebraic immunity, one must first understand some basic terminology. Most cryptographic systems are based on Boolean functions. A Boolean function is simply a mapping from the vector space of *n*-tuples of bits to the field of two elements. Simply, a Boolean function takes a collection of bits and transforms them into a different form, and it is through these transformations that cryptosystems operate.

Algebraic immunity is a measure of a particular Boolean function's resistance against algebraic attacks. An algebraic attack is accomplished by utilizing low degree functions, called annihilators, to reduce the complexity of a given Boolean function to a form that is closer to linear. Linear systems are simple to solve, and linearizing a system is the first step in the process of breaking a cryptosystem when performing an algebraic attack.

Algebraic immunity is the lowest degree of an annihilator of a function or its complement. Degree is the maximum number of unique variables in a term for a function. For example, the function $f = x_1 + x_2 x_3$ has a degree of two because there is a maximum of two unique variables in any term in the function. The complement of a function is obtained by changing all of the 1s to 0s in a function and vice versa.

Many of the computations of AI in this work were carried out on the SRC-6. The SRC-6 is a reconfigurable computer that contains 10 FPGAs, each of which can be specifically programmed to carry out the desired computation. The most complex computation in this work required only 10% of the resources of a single FPGA, demonstrating the processing power of the SRC-6.

There are many methods of computing AI. The first method utilized in this work was a brute force method which applied all possible inputs to the function being tested to

determine the annihilators. This method is thorough, but it is not efficient. The algebraic immunity was successfully computed for all functions with four variables, but it operated so slowly that it took nearly a minute to compute the algebraic immunity for some functions of five variables. Since there are $2^{2^n} = 2^{2^5} = 4{,}294{,}967{,}296$ functions in $n = 5$ variables, a more efficient method of computing AI was required.

This led to the development of the simultaneous equation algorithm. This algorithm first builds a matrix that represents all possible annihilators of a given Boolean function. It then quickly determines the lowest degree annihilator of the function and its complement, which is the AI of the function being tested.

Some mathematical shortcuts are responsible for the speed of the simultaneous equation algorithm. First, the algorithm begins placing the matrix into reduced row echelon form. A matrix in this form has only a single one in each column, and every leading one, or first one in a row, is in a row higher than every leading one to its right. This results in a matrix where the ones are restricted to the upper triangle of two triangles (if the matrix were sliced in half diagonally from the upper left to the lower right, there would be two triangles).

The algorithm reduces the matrix to reduced row echelon form by searching each column, starting at the left, for a one, and then adding that row to every other row that has a one in the column being searched. This ensures that the row where a one was found is the only row in that column that is nonzero.

If the algorithm discovers an empty column, it stops searching immediately. An empty column represents a free variable and signifies that an annihilator has been found. Similarly, once all the columns for a particular degree have been checked, the algorithm stops to determine if an annihilator exists. Also, the algorithm does not search for annihilators of the maximum lowest degree, as such an annihilator is guaranteed to exist if lower degree annihilators do not exist.

Utilizing this algorithm, we performed the first known enumeration of AI for all five variable functions. There were 7,666,550 functions with an AI of one, which

correlates with the proven value. There were 4,089,535,624 functions with an AI of two, and there were 197,765,120 functions with an AI of three. There are no known outside sources of comparison at this time.

The algorithm was coded in C and an enumeration of all five variable functions was performed using a conventional processor. The results were identical, although it took the conventional processor 4.86 times as long to achieve the results.

After these successes, the algorithm was altered to perform a set of Monte Carlo trials for $n = 6$. Monte Carlo trials are random trials performed to test a property on a group that is too large to fully enumerate. They allow the distribution of the group to be estimated so long as the trials are sufficiently random.

To ensure randomness, the Mersenne Twister pseudorandom number generator was utilized. This generator has known good properties that make it suitable for use in Monte Carlo trials.

The calculated distribution for six variable functions showed that more than 90% of the functions have the maximum possible AI of three. Only a very small percentage of functions had an AI of one, which matches the calculated value for that number.

For six variable functions, the C code outperformed the SRC-6, computing the AI for functions 46% faster. This was primarily due to the difficulty of implementing a pseudorandom function on the SRC-6, which runs Verilog code. Instead of generating pseudorandom numbers in Verilog, they were generated in C and memory transferred to the SRC-6 FPGA. These memory transfers are suspected to be the cause of the slowdown.

While performing the initial work for the brute force algorithm, we discovered a need for a faster transeunt triangle. A transeunt triangle is a collection of XOR gates in a triangular configuration, with the two gates that are adjacent in one level of the triangle providing the input to the same gate in the next level. The transeunt triangle alters a Boolean function between truth table form and algebraic normal form. Truth table form is a form that specifies what the output of the Boolean function is for a given set of inputs. Algebraic normal form specifies which terms are present in the Boolean function.

Each of these forms is useful for determining different properties regarding a Boolean function. For this work, the Boolean function is input in truth table form as that provides an easier method of computing annihilators. The annihilators are output in truth table form and must be converted to algebraic normal form so that their degree can be determined.

The complete transeunt triangle was too slow to meet the timing requirements necessary to implement the brute force algorithm, so the first known Verilog implementation of the reduced transeunt triangle was developed. The reduced transeunt triangle only requires five gate delays to convert a five-variable Boolean function between forms, while a complete transeunt triangle requires 25 gate delays for this conversion. This time savings was critical in allowing the brute force algorithm to function properly.

An efficient algorithm for computing algebraic immunity was developed and the first known enumeration of AI for all functions in five variables was completed in this research. Further effort is required to extend this work to functions with more variables in an effort to secure cryptosystems that are used in real-world applications.

ACKNOWLEDGMENTS

First I would like to thank God for getting me here, despite my occasional best efforts otherwise.

My wife, Kimberly, is my Rock, and I would never have finished without her loving support.

My children, Cody, Krista, James, Aleah, and Olivia, provided a constant source of inspiration and an occasional source of distraction while I was completing this.

I thank my Mom and Dad for everything, really, but also for making sure I wasn't procrastinating and for coming out when I needed them. I also thank my sister, Sarah, for always being good competition, and my brother, Chris, for believing in me.

I thank my mother-in-law, Denise, and her husband, Rodney, for their support during some critical times.

Without the tireless efforts of Dr. Jon Butler and Dr. Pantelimon Stanica, this thesis would have never been finished. They provided aid to me from literally almost every corner of the world, and were as excited about my successes as I was. Dr. Butler always knew the right question to ask to get me past a roadblock. I separately thank Dr. Stanica for securing my Erdõs number of 3.

I thank Dan Zulaica for his valuable support with the SRC-6.

I thank CDR Schoolsky for his assistance with non-academic affairs that allowed me to better focus on the academics.

I thank Alice Lee for taking care of the administrative items and making the process as painless as possible.

Finally, I thank all of the great Naval Postgraduate School instructors and assistants not already mentioned for making sure I left here at least a little more knowledgeable than when I came.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. OBJECTIVE

Algebraic immunity (AI) is an important cryptographic property. A Boolean function that has a low value for this parameter is provably insecure. Prior to this work with the SCR-6, the distribution of AI among Boolean functions was only known through $n = 4$. The objective of this work is to exhaustively determine the distribution of AI for all functions through $n = 5$ and to provide a method to test specific groups of Boolean functions for larger values of n. An exhaustive search of Boolean functions for $n = 6$ and beyond exceeds the capabilities of modern hardware. The methods utilized in this work, such as the first Verilog implementation of a reduced transeunt triangle, can allow entire groups of Boolean functions to be tested quickly for all important cryptographic properties.

B. BACKGROUND

Algebraic attacks involve manipulating high degree Boolean functions through multiplication with lower degree functions to create a system of equations that is more easily solved. These attacks were first discussed a decade ago [1, 2]. Since that time, they have continuously evolved, with more efficient algorithms appearing that are also closely targeted at specific types of encryption methodologies [3, 4].

The use of encryption has been growing steadily in all sectors. It is important for consumers, businesses, and governments [5]. Many symmetric encryption techniques are based on the use of Boolean functions. Strong encryption requires choosing these functions such that they have desirable cryptographic properties. The SRC-6 has been previously used to quantify other properties, such as correlation immunity [6] and bentness [7–9]. Algebraic immunity has not yet been addressed.

Determining all cryptographic properties for a given function or class of functions allows a more informed decision to be made regarding the viability of those functions to secure communications. The heavy reliance of the Department of Defense on encryption

to conduct operations at all levels requires a continued effort to improve communications security [10, 11]. An efficient method for evaluating encryption standards is vital.

C. METHOD

A Boolean function is specified by its truth table (TT) form or algebraic normal form (ANF). It is trivial to enumerate the truth tables of all Boolean functions for a given number of variables. This is performed by counting from 0 to $2^{2^n}-1$, where n is the number of variables in the Boolean functions of interest.

As each Boolean function is enumerated, its AI is tested by finding the smallest degree annihilator, where an annihilator is any function that reduces the original Boolean function to zero when the two are multiplied (bitwise AND) together. Various algorithms are utilized to find low degree annihilators of the function under test (FUT) and its complement.

Once the FUT has been tested with all possible annihilators, the degree of the lowest degree annihilator is the AI. By testing all functions of specific degrees we can determine the distribution of AI. Monte Carlo methods allow us to estimate the distribution for higher degrees that cannot be exhaustively tested.

Some testing is performed on a conventional processor using algorithms written in C, but most testing is performed on the SRC-6. The SRC-6 has ten field programmable gate arrays (FPGA) that each operate at 100 MHz. While this is an order of magnitude slower than conventional processors, the machine's power lies in its ability to be specifically programmed for a given task. In our case, a circuit was designed that takes in the FUT and performs all tasks required to calculate the AI. This allows functions to be tested more rapidly than with a conventional processor.

D. RELATED WORK

There have been many theses researching various cryptographic properties utilizing the programmability of the SRC-6 [6–9]. The properties covered in those works relate with AI because a strong cryptographic function requires all properties to exhibit

desirable characteristics. Significant work is being performed on algebraic immunity itself, with most work focusing on new algorithms for determining annihilators and for computing AI [1–4].

E. THESIS OUTLINE

The thesis is introduced in Chapter I. An introduction to linear algebra is provided in Chapter II. The concept of AI and its impact on the cryptographic viability of a Boolean function is discussed in Chapter III. The reduced transeunt triangle is discussed in Chapter IV. The algorithms used for computing AI are discussed in Chapter V. Results are discussed in Chapter VI. Conclusion and recommendations for future work are discussed in Chapter VII. All SRC-6 code utilized for this work is contained in Appendix A. All C code utilized for this work is contained in Appendix B.

THIS PAGE INTENTIONALLY LEFT BLANK

II. LINEAR ALGEBRA INTRODUCTION

A. MATRICES

1. System of Linear Equations

A matrix is a representation of a system of linear equations.

Example 1: The equations $2x+4y=26$ and $3x+2y=19$ can be represented by the coefficient matrix and the solution matrix

$$\begin{pmatrix} 2 & 4 \\ 3 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 26 \\ 19 \end{pmatrix}. \tag{1}$$

2. Augmented Matrix

An augmented matrix is created by combining the coefficient matrix with the solution matrix to produce a single matrix.

Example 2: The matrices listed in Example 1 can be combined to form the augmented matrix

$$\left(\begin{array}{cc|c} 2 & 4 & 26 \\ 3 & 2 & 19 \end{array}\right). \tag{2}$$

Augmented matrices combine all terms of the equations allowing the system of equations to be solved for any unknown values. If all solutions are zero, manipulations of the matrix have no impact on the solution, and the augmented matrix may be discarded in favor of using a simple coefficient matrix.

3. Matrix Equivalency

Two matrices *A* and *B* are considered equivalent if there is an invertible *m*-by-*m* matrix *C* and an invertible *n*-by-*n* matrix *D* such that $A = C^{-1} \cdot B \cdot D$. This means that the two matrices *A* and *B* represent the same linear transformation.

B. ELEMENTARY ROW OPERATIONS

There are three elementary row operations that can be performed on a matrix that produce an equivalent matrix. These row operations are utilized to alter the matrix into a different form, such as row echelon form or reduced row echelon form, in order to reduce any additional computation required to solve the system of equations.

1. Interchange two rows.

Any two rows of a matrix can be interchanged to yield a second equivalent matrix.

Example 3: The two matrices *A* and *B* are equivalent because matrix *B* is formed by interchanging rows one and two of matrix *A*:

$$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix} \quad B=\begin{pmatrix}3 & 4\\ 1 & 2\end{pmatrix}. \tag{3}$$

2. Multiply any row by a nonzero number.

If any row of a matrix is multiplied by a nonzero number, the result is an equivalent matrix.

Example 4: The two matrices *A* and *B* are equivalent because matrix *B* is formed by multiplying row one of matrix *A* by 3:

$$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix} \quad B=\begin{pmatrix}3 & 6\\ 3 & 4\end{pmatrix}. \tag{4}$$

3. Multiply any row by a nonzero number and add the result to another row.

Any row of a matrix can be multiplied by a nonzero number and then added to a second row, replacing the original contents of the second row. This operation is typically used to cancel certain terms in a row.

Example 5: The two matrices *A* and *B* are equivalent because matrix *B* is formed by multiplying row one of matrix *A* by -3 and adding the result to matrix *A*:

$$A=\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix} \quad B=\begin{pmatrix}1 & 2\\ 0 & -2\end{pmatrix}. \tag{5}$$

C. ROW ECHELON FORM

A matrix is in row echelon form if it satisfies the following three criteria:

1. **The leading coefficient in a nonzero row is 1.**
2. **Any row with nonzero coefficients has fewer leading zeros than all rows below it.**
3. **Rows with all zeros are below all rows with nonzero coefficients.**

Example 6: These matrices are in row echelon form:

$$\begin{pmatrix}1 & 3 & 7\\ 0 & 1 & 2\\ 0 & 0 & 1\end{pmatrix} \quad \begin{pmatrix}1 & 2 & 4\\ 0 & 0 & 1\\ 0 & 0 & 0\end{pmatrix}. \tag{6}$$

Transforming a matrix into row echelon form allows the corresponding linear system to be solved using back substitution. Back substitution is a method of solving the lowest equation, or the lowest nonzero row of the matrix, and then substituting the result into the next higher equation, i.e., the next higher row. This process is repeated until the system of equations is solved completely.

D. REDUCED ROW ECHELON FORM

Reduced row echelon form is an extension of row echelon form. After a matrix is in row echelon form, it is transformed to reduced row echelon form by eliminating all nonzero coefficients in each column containing a leading coefficient. When a matrix is in reduced row echelon form, the leading coefficient for each row is also the only nonzero coefficient in its column. Placing a matrix in reduced row echelon form is an essential step in many algorithms utilized to compute algebraic immunity.

Example 7: These matrices are in reduced row echelon form:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}. \tag{7}$$

Linear algebra techniques are used to solve simultaneous equations in many areas of mathematics. Forming a matrix that represents a system of equations and solving it by placing it in reduced row echelon form is critical in many algorithms used to compute algebraic immunity.

III. ALGEBRAIC IMMUNITY

A. DEFINITIONS

1. Group

A **group** G is a set, or collection of objects, combined with an operation, denoted *, which together satisfy the group axioms:

a. Closure: $a*b \in G \, \forall a,b \in G$;

b. Identity: $\exists e \in G \, s.t \, \forall a \in G, e*a = a*e = a$;

c. Associativity: $(a*b)*c = a*(b*c) \, \forall a,b,c \in G$; and

d. Invertibility: $\forall a \in G, \exists (-a) \in G \, s.t. \, a*(-a) = e$.

2. Abelian Group

An **Abelian group** G is a group whose operation is Abelian, or commutative, meaning that $\forall a,b \in G, a*b = b*a$.

3. Ring

A **ring** is a set along with two operations on the set that satisfies the ring axioms:

a. The set is an Abelian group under addition,

b. The set is closed under multiplication,

c. The set is Associative under multiplication, and

d. The set is Distributive.

4. Field

A **field** is a ring with the following properties:

a. The ring is commutative,

b. The ring has unity; i.e., the ring possesses an identity element with respect to multiplication, and

c. Every nonzero element in the ring is a unit [12].

Example 8: The symbol $\mathbb{F}_2$ is the Galois field over two elements. It has an addition operation which functions as the XOR of two bits and a multiplication operation which functions as the AND of two bits. The TT for these operations is shown in Table 1 and Table 2.

Table 1. TT for addition (XOR) modulo-2.

a	*b*	$x = a+b$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2. TT for multiplication (AND) modulo-2.

a	*b*	$x = a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

5. Vector Space

A **vector space** over a scalar set is a set *V* which, given any vectors **x**, **y**, and **z** in *V* and any scalars *a* and *b*, satisfies the following properties:

a. Closure under addition: $\mathbf{x}+\mathbf{y} \in V$,

b. Closure under multiplication: $a \cdot \mathbf{x} \in V$,

c. Commutative: $\mathbf{x}+\mathbf{y} = \mathbf{y}+\mathbf{x}$,

d. Inverse: $\forall \mathbf{x} \in V \; \exists (-\mathbf{x}) \in V \; s.t. \; \mathbf{x}+(-\mathbf{x}) = \mathbf{0}$,

e. Additive Identity: $\exists 0 \in V \; s.t. \mathbf{x} + 0 = 0 + \mathbf{x} = \mathbf{x}$,

f. Multiplicative Identity: $\exists 1 \in V \; s.t. 1 \cdot \mathbf{x} = \mathbf{x} \cdot 1 = \mathbf{x}$,

g. Additive Associativity: $(\mathbf{x} + \mathbf{y}) + \mathbf{z} = \mathbf{x} + (\mathbf{y} + \mathbf{z})$,

h. Multiplicative Associativity: $(a \cdot b)\mathbf{x} = a(b\mathbf{x})$,

i. Scalar Distributivity: $a(\mathbf{x} + \mathbf{y}) = a\mathbf{x} + a\mathbf{y}$, and

j. Vector Distributivity: $\mathbf{x}(a + b) = \mathbf{x}a + \mathbf{x}b$.

Example 9: The symbol $\mathbb{V}_n$, which can also be represented as $\mathbb{Z}_2^n$, is the vector space consisting of all *n*-tuples of bits. The first operation for $\mathbb{V}_n$ is vector addition: $\mathbf{v_1} + \mathbf{v_2} = \mathbf{v_3}$, where $\mathbf{v_1} = (a_1, \cdots, a_n)$, $\mathbf{v_2} = (b_1, \cdots, b_n)$, and $\mathbf{v}_3 = (a_1 + b_1, \cdots, a_n + b_n)$. The second operation is scalar multiplication: $\alpha \cdot \mathbf{v_1} = \alpha \mathbf{v_1}$, where α is any scalar and $\mathbf{v_1} = (a_1, \cdots, a_n)$.

6. Boolean Function

A **Boolean function** is a function which maps variables from a vector space $\mathbb{V}_n$ to the field $\mathbb{F}_2$, where *n* is the number of variables. There are 2^{2^n} distinct Boolean functions for each value of *n*.

7. Degree

The **degree** of a Boolean function is the largest number of variables that appear in a single term of the function.

Example 10: The function $f = x_1 + x_2 + x_2x_3$ is of degree two because there are two variables in the x_2x_3 term.

B. ANNIHILATORS

An **annihilator** is a nonzero function *g* such that, for the given function *f*, $g \cdot f = 0$. The function *g* is said to be an annihilator of *f*, or one can say that *g* annihilates *f*.

Example 11: $f(x) = x_1 + x_1x_2$

$g(x) = x_2$

$f \cdot g = (x_1 + x_1x_2) \cdot (x_2)$

$= (x_1x_2 + x_1x_2x_2)$

$= (x_1x_2 + x_1x_2)$

$= 0$

Thus, *g* annihilates *f*.

C. ALGEBRAIC IMMUNITY

Algebraic immunity is the measure of a Boolean function's resistance to an algebraic attack. Specifically, the AI of a Boolean function is the lowest degree of any annihilator of the function or its complement.

Example 12: The AI of $f(x) = x_1 + x_1x_2$ is one because it is annihilated by the degree one function $g(x) = x_2$, as shown in Example 11, and it cannot have a degree zero annihilator.

1. Range of Algebraic Immunity

For a given number of variables, there is a known range of possible AI values for the functions with that number of variables. Only the constant functions, whose truth tables are all ones or all zeros, have an AI of zero. All other functions have a lower bound of one for AI and an upper bound of $\lceil n/2 \rceil$, where *n* is the number of variables for the function [13].

Knowledge of the range of possible values for AI greatly simplifies the search for the lowest degree annihilator of a function. If no annihilators are found through $\lceil n/2 \rceil - 1$, then it is clear that the lowest degree annihilator is $\lceil n/2 \rceil$, as the degree of the annihilator can be no higher. Each increasing degree inserts a large number of potential annihilators, so utilizing the upper bound when determining AI dramatically speeds up the process.

2. Symmetry of AI

The AI of a function and its complement are always the same, as the AI is the lowest degree annihilator of either function. If a degree one annihilator is found for a function, there is no need to analyze its complement, as there can be no lower degree. The converse is not true. Finding an annihilator of degree $\lceil n/2 \rceil$ for a function does not alleviate the need to check its complement, as the AI of both functions is ultimately the lowest degree annihilator of either. Use of this symmetry can provide a speedup for an AI algorithm by eliminating unnecessary checks once a degree one annihilator has been found.

Some algorithm used for computing AI return the resultant annihilator in TT form. This form does not immediately provide the degree of the annihilator. Transforming the annihilator to ANF is required to determine the degree. The transformation between TT form and ANF is accomplished using a transeunt triangle.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. REDUCED TRANSEUNT TRIANGLE

In this chapter *n* represents the number of inputs into a given transeunt triangle and not the number of variables associated with a particular Boolean function.

A. THE COMPLETE TRANSEUNT TRIANGLE

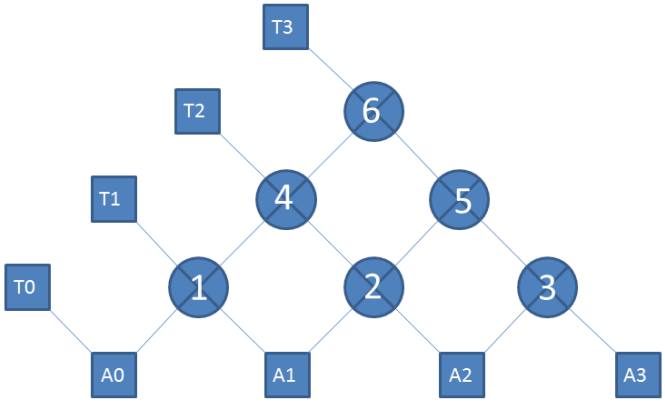

Figure 1. Complete four-input transeunt triangle.

A transeunt triangle is a collection of exclusive-or (XOR) gates that transform a binary input into another representation. A four-input complete transeunt triangle is shown in Figure 1. Their primary function in cryptographic research is to take as input a Boolean function in TT form and provide its ANF as output, and vice versa. It has already been proven that the transeunt triangle transforms a Boolean input between these two forms [9].

The reason for transforming a function between the two forms is that TT form is more useful in some instances, while the ANF is advantageous in other cases. For example, one of the algorithms utilized to calculate AI in this work requires the input to be in TT form, but to determine the degree of the computed annihilator requires the output to be in ANF. The transeunt triangle allows this conversion to be made so that both forms can be utilized inside the same algorithm.

While the complete transeunt triangle is effective at converting between a TT form and its ANF, it is not the most efficient method of performing this conversion.

There are intermediate values computed throughout the transeunt triangle that are not necessary in the computation of one form from the other. These unnecessary computations result in increased resource utilization and latency. While the additional amount of logic and delay is negligible for a four-input transeunt triangle, the increases become overwhelming for significantly larger transeunt triangles. This resulted in the development of a more efficient method of converting between TT form and ANF.

B. REDUCED TRANSEUNT TRIANGLE DEVELOPMENT

Looking at the simple four-input complete transeunt triangle in Figure 1, unnecessary gates can easily be seen. The output T2 is produced by an exclusive-or of A0 and A1. The output T3 is produced by an exclusive-or of T2 with the exclusive-or of A1 and A2. This results in:

$$T3 = (A0 \oplus A1) \oplus (A1 \oplus A2) = A0 \oplus A2. \tag{8}$$

This result can be produced by instead performing an exclusive-or of A0 and A2 directly, allowing gate 2 in Figure 1 to be removed. Similar logic allows the removal of gate 5 so that the output of gates 1 and 3 go directly to gate 6. Removing these unnecessary gates produces the reduced transeunt triangle shown in Figure 2.

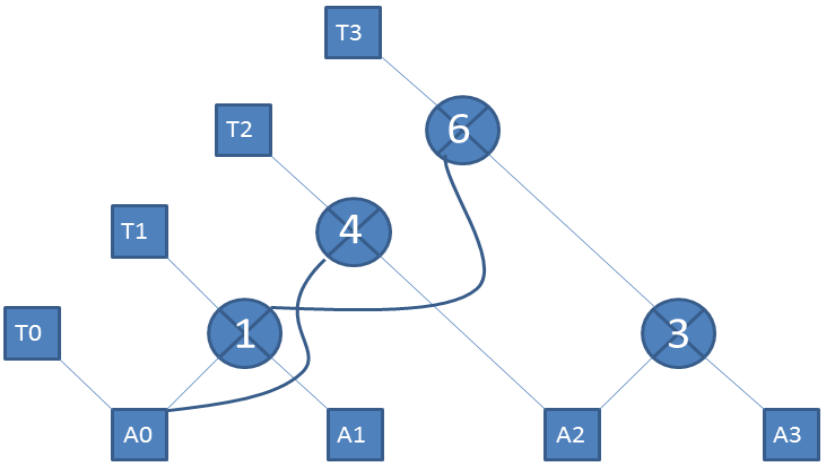

Figure 2. Four-input reduced transeunt triangle.

The basic four-input reduced transeunt triangle can be extended to an arbitrarily larger reduced transeunt triangle. The extension of the reduced transeunt triangle to accept $2n$ inputs requires two n-input reduced transeunt triangles, along with an additional n XOR gates. This is demonstrated for an eight-input reduced transeunt triangle in Figure 3.

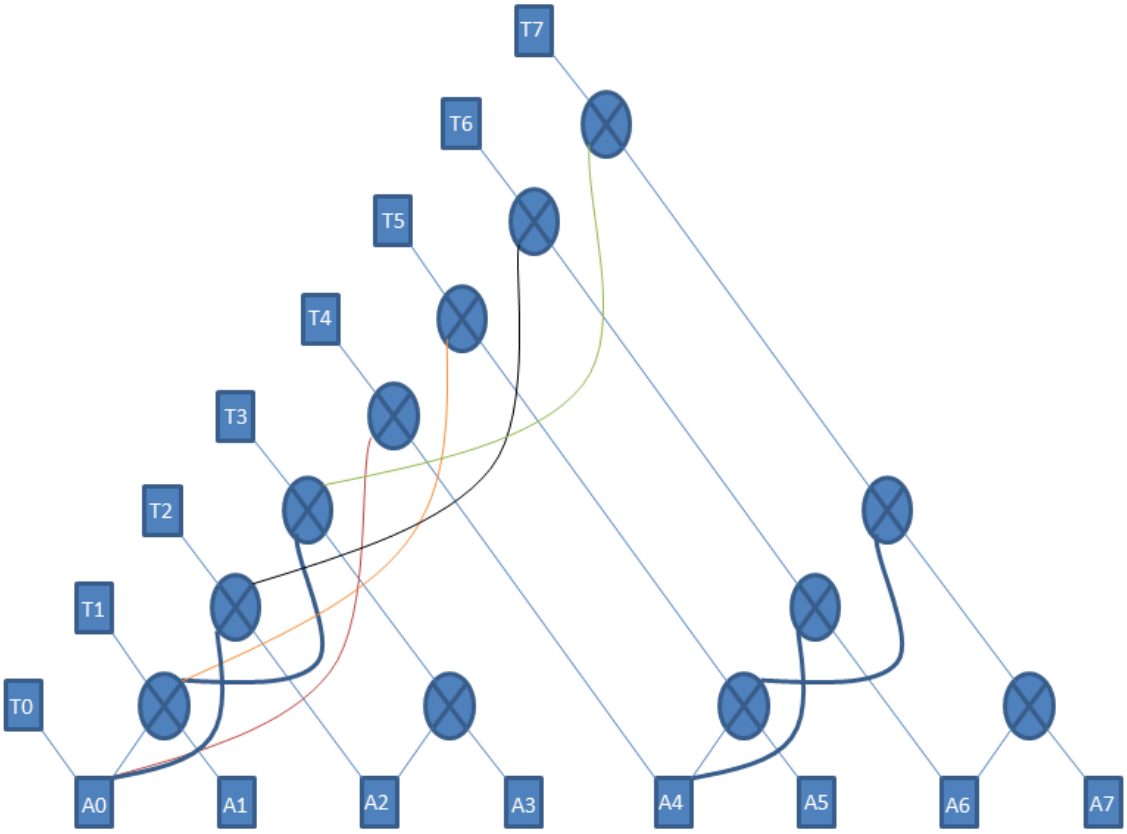

Figure 3. Eight-input reduced transeunt triangle.

The reduced transeunt triangle utilized in this work was independently discovered. There are other examples in the literature which offer similar benefits and that have a similar structure, but there are differences in their specific layout and the manner in which they are extended to cover a larger number of inputs [14]. This reduced transeunt triangle has a simple recursive nature that more simply shows its extension to accept a larger number of inputs than other variations.

C. REDUCED TRANSEUNT TRIANGLE EQUIVALENCY

In order to replace the complete transeunt triangle with the reduced transeunt triangle, we must establish that the two produce equivalent outputs.

Theorem IV.C.1: *The reduced transeunt triangle produces output equivalent to the output produced by the complete transeunt triangle.*

Proof:

The proof is by induction, beginning with $n = 4$, the smallest case for which the reduced transeunt triangle removes XOR gates from the complete transeunt triangle.

Applying inputs A0, A1, A2, and A3 to the complete transeunt triangle in Figure 1 results in the following equations for the outputs:

$$\begin{aligned} T0 &= A0 \\ T1 &= A0 \oplus A1 \\ T2 &= (A0 \oplus A1) \oplus (A1 \oplus A2) = A0 \oplus A2 \\ T3 &= ((A0 \oplus A1) \oplus (A1 \oplus A2)) \oplus ((A1 \oplus A2) \oplus (A2 \oplus A3)) \\ &= A0 \oplus A1 \oplus A2 \oplus A3 \end{aligned} \tag{9}$$

Applying these same inputs to the reduced transeunt triangle in Figure 2 results in the following equations for the outputs:

$$\begin{aligned} T0 &= A0 \\ T1 &= A0 \oplus A1 \\ T2 &= A0 \oplus A2 \\ T3 &= (A0 \oplus A1) \oplus (A2 \oplus A3) = A0 \oplus A1 \oplus A2 \oplus A3 \end{aligned} \tag{10}$$

This shows that the complete and reduced transeunt triangles are equivalent for $n = 4$.

Next, we assume that the complete and reduced transeunt triangles are equivalent for *n* and show that this must be true for 2*n*. We place two *n*-input triangles together and connect their outputs to a string of XOR gates that forms the top left of a larger triangle as illustrated in Figure 4.

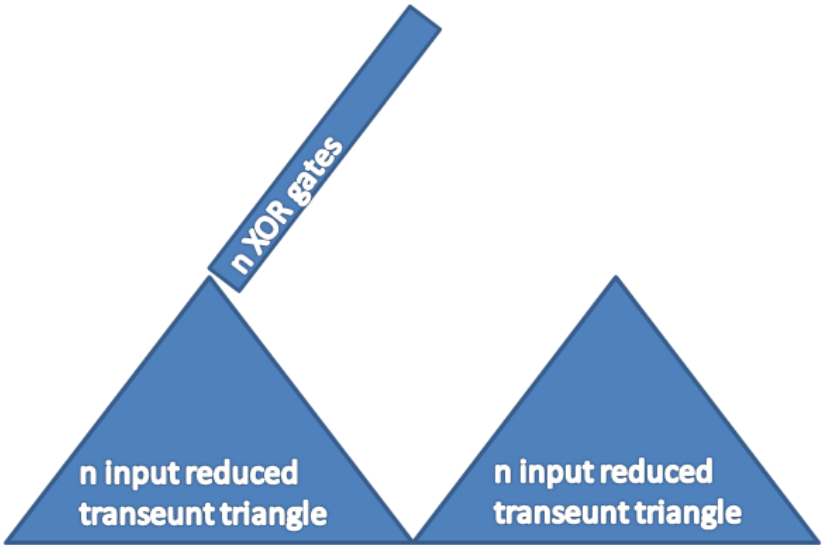

Figure 4. Combination of *n*-input reduced transeunt triangles and XOR gates.

This allows us to see how the inputs are combined to form the overall output. The top output of each *n*-input reduced transeunt triangle is the XOR of all its inputs. Since each of these receives half of the 2*n* overall inputs, combining them with via an XOR gate provides the desired overall output of the XOR of all 2*n* inputs. Similarly, the 2nd input from the top of each smaller triangle is the XOR of every other input for each half of the 2*n* inputs, and so the XOR of the outputs from the two smaller triangles results in the overall output being the XOR of every other term for all 2*n* inputs. Similar logic combines each output of the two separate *n*-input reduced transeunt triangles via an XOR gate to produce the desired overall output for the 2*n*-input reduced transeunt triangle.

Q.E.D.

D. REDUCED TRANSEUNT TRIANGLE ADVANTAGES

The reduced transeunt triangle offers several advantages over a complete transeunt triangle, the first of which is greatly improved resource efficiency. The number of gates required to make a complete transeunt triangle with *n*-inputs is $\sum_{i=1}^{n-1} i = n(n-1)/2$, while the number of gates for a reduced transeunt triangle with *n* inputs is defined for *n*

equal to powers of two by the recursion $a_n = 2 \cdot a_{n/2} + n/2, a_2 = 1$ [14]. The numbers of gates for a complete transeunt triangle grows at a rate of $n(n-1)/2$ while the number required for a reduced transeunt triangle only grows at a rate of $n \cdot \ln(n)/2$ [14]. This results in a significant reduction in the required number of gates as the number of inputs increases, as shown in Table 3.

Table 3. Complete and reduced transeunt triangle gate comparison.

Inputs	Gates (complete transeunt triangle)	Gates (reduced transeunt triangle)	Percent reduction
2	1	1	0%
4	6	4	33.33%
8	28	12	57.14%
16	120	32	73.33%
32	496	80	83.87%
64	2016	192	90.48%
128	8128	448	94.49%
256	32640	1024	96.86%

The reduction in gate delay afforded by utilizing the reduced transeunt triangle is as important as the reduction in the number of gates. For a complete transeunt triangle, the signal must propagate from the inputs to the very top of the triangle before the output is ready. This results in a delay of $n-1$, where n is the number of inputs. The reduced transeunt triangle only requires one additional gate delay for each doubling of inputs, so delay increases logarithmically, a significant improvement. When working with a typical number of inputs for a cryptographic function, this speedup is the difference between an efficient circuit and one that is not. The number of gate delays for the complete and reduced transeunt triangles and the percent speedup achieved at various input numbers is shown in Table 4.

Table 4. Complete and reduced transeunt triangle delay comparison.

Inputs	Delay (complete transeunt triangle)	Delay (reduced transeunt triangle)	Percent speedup
2	1	1	0%
4	3	2	50%
8	7	3	133%
16	15	4	275%
32	31	5	520%
64	63	6	950%
128	127	7	1714%
256	255	8	3087%

The final major advantage of the reduced transeunt triangle is that its recursive nature allows it to be easily produced in Verilog. While the complete transeunt triangle has been produced in Verilog in other works, the code to do so is not as clean and easy to follow as that for the reduced transeunt triangle [9]. As each doubling of inputs simply adds one level to the recursion, the reduced transeunt triangle can be made to an arbitrary size relatively simply in any programming language.

The advantages of the reduced transeunt triangle allow it to function in circuits at a speed not possible using a complete transeunt triangle. This speedup is critical in determining the degree of an annihilator in the brute force algorithm for computing algebraic immunity.

THIS PAGE INTENTIONALLY LEFT BLANK

V. ALGEBRAIC IMMUNITY ALGORITHMS

A. INTRODUCTION

Throughout the course of this work, several algorithms were developed to efficiently calculate AI. The algorithms varied in complexity and capability, with each succeeding algorithm having increased capability over the previous at the cost of increased complexity. The final algorithm is the most capable and best lends itself to future work, but there are interesting results from each algorithm.

B. BRUTE FORCE COMBINATORIAL ALGORITHM

1. Overview

The brute force combinatorial algorithm provides a mechanical processing of the input FUT to calculate the AI. It is best understood looking at the top level view show in Figure 5.

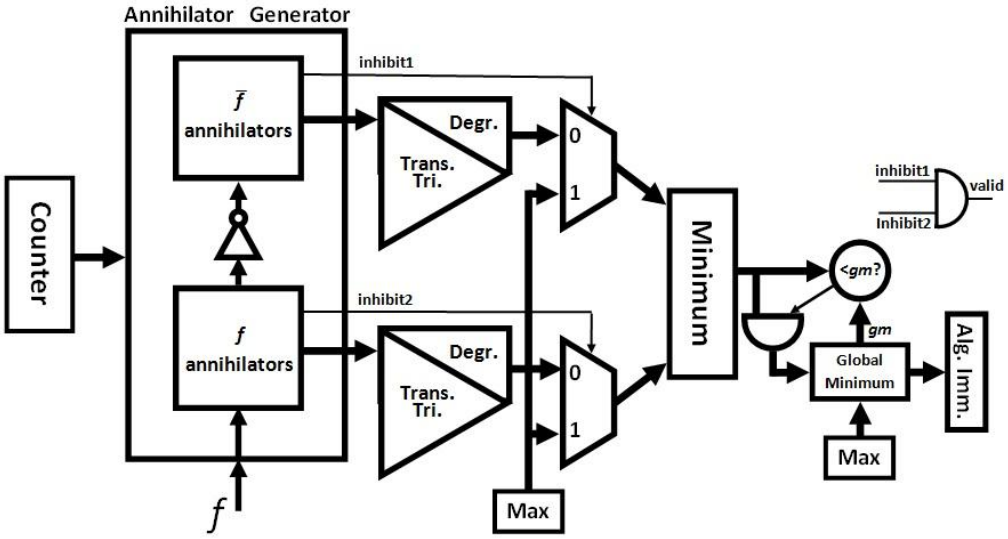

Figure 5. Brute force combinatorial algorithm top-level view.

Looking at the figure, the FUT f is applied in TT form to the annihilator generator. The counter successively applies inputs to the function and its complement. It does this by starting at one and incrementing every clock cycle. It must start at one,

because zero is an annihilator for all functions. The counter outputs are applied to the values in f that have a one in the TT to produce annihilators of f. Similarly for $\overline{f}$, the complement of f, the counter outputs are applied to the values in f that have a zero in the TT to produce the annihilators of $\overline{f}$.

After the annihilators are produced in the annihilator generator, they are output to the transeunt triangle where they are converted to an ANF. Next, their degree is determined, and this degree is compared with the global minimum to see if a new lowest degree annihilator has been found. The inhibit signals allow the output of the annihilator generator for either f or $\overline{f}$ to be ignored since, for unbalanced functions, either the function or its complement will have all possible annihilators exhausted first due to the mismatch in the number of 1s and 0s in the TT. When one output is inhibited, the maximum possible AI is substituted for the output of that annihilator generator.

Once both annihilator generators have exhausted all possible annihilators, as signaled by the inhibit signal, the global minimum value represents the AI of the FUT and is output to the controlling function. The next function can then be tested in a similar manner.

2. Advantages

The first advantage of this algorithm is that it operates very quickly due to its combinatorial nature. A requirement is that all operations either complete within one clock cycle or can be pipelined so that a portion completes within a clock cycle.

Although this implementation does not track this data, it is a relatively simple modification to store the annihilators that are computed in the annihilator generator. Knowing all annihilators provides a considerable benefit when performing an algebraic attack on a Boolean function [15].

3. Disadvantages

The first disadvantage of the brute force combinatorial algorithm is that its performance suffers for unbalanced functions. When a function is balanced, meaning that its truth table has the same number of ones and zeros, the performance of this

algorithm is optimal. As a function becomes more unbalanced, one half of the annihilator generator becomes idle since the other half is responsible for processing the majority of the possible annihilators, lowering efficiency.

This algorithm does not operate quickly enough to run on the SRC-6 for $n \geq 5$. Without an undue effort spent pipelining the design, this algorithm would not run for values of *n* larger than three, and so a new algorithm was developed.

C. BRUTE FORCE STATE MACHINE ALGORITHM

1. Overview

The requirement for increased speed resulted in the development of a brute force state machine algorithm. This algorithm executes in a manner very similar to the combinatorial algorithm, but it utilizes states to shift some of the processing to different clock cycles in order to execute within the clock period of the SRC-6. A state machine diagram is shown in Figure 6.

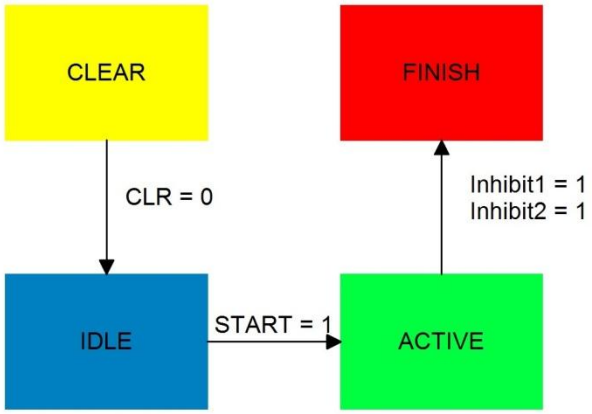

Figure 6. Brute force state machine algorithm state diagram.

The CLEAR state initializes some of the variables that are used in the operation of the algorithm, and the IDLE state continues with initialization once the clear signal has been removed. Once the start signal is received, the active state begins counting and applying annihilators to the FUT, just as in the combinatorial algorithm previously

discussed. Once both inhibit signals are received, the algorithm exits and returns to the controlling function, delivering the calculated AI.

This algorithm relies on the reduced transeunt triangle to accomplish all processing during a clock cycle, and the development of this algorithm was directly responsible for the development of the reduced transeunt triangle. Without the efficiency of the reduced transeunt triangle this algorithm would too slow to complete all required calculations within one clock period.

2. Advantages

The primary advantage of the state machine algorithm over the pure combinatorial algorithm is that this algorithm will successfully run for $n = 4$. By moving the initialization to various states, the ACTIVE state is capable of performing all functions during each clock cycle to iterate through the annihilators and search for the lowest value.

Another advantage of this algorithm is that it is easily parameterized. This makes the process of moving between different values of *n* very simple, requiring the alteration of a single parameter to accomplish all required changes. Other algorithms can prove more tedious to adapt to varying values of *n*.

3. Disadvantages

This brute force state machine algorithm suffers from the same inefficiencies due to unbalanced functions as the combinatorial algorithm described previously. The vast majority of the processing time to enumerate all functions for $n = 4$ is spent on unbalanced functions.

The major disadvantage of this algorithm is that the runtime precludes its use to enumerate all Boolean functions for $n = 5$. When performing the first trial runs for small ranges at $n = 5$, it was noted that some individual functions required nearly four billion clock cycles. This is because, for unbalanced functions, the counter may need to run as high as 2^{31} to process a single function. It is this failure of the brute force state machine algorithm that required the development of a more efficient method to compute AI.

D. SIMULTANEOUS EQUATION ALGORITHM

1. Overview

The simultaneous equation algorithm operates in a manner similar to most algorithms that are being pursued by other sources, which are all variations on solving the simultaneous equations that are built from the TT of the FUT [2–4, 13, 15, 16]. This is the first known implementation using Verilog on an FPGA.

This algorithm starts by using the TT for the FUT to populate a matrix we call the annihilator matrix. This matrix represents the terms that exist in all possible annihilators of the FUT. The matrix is produced by examining each bit which is a one in the TT. Every bit has a one-to-one correspondence with a row in the matrix because that row represents the ANF of the minterm of a particular bit in the TT. For instance, the 4^{th} row in the matrix corresponds to the ANF of the minterm for the 4^{th} bit in the TT if both are numbered starting at zero. Each annihilator matrix is unique because the TT is unique for each Boolean function.

For example, given $n = 5$, there are 32 unique rows which can be part of the matrix, producing a 32 by 32 matrix. Each row where the TT for the FUT has a one is filled with the default values from this complete matrix, and each row where the TT has a zero is filled with all zero. Solving the annihilator matrix provides all possible annihilators for the FUT. Computation of the AI does not require the matrix to be fully solved, but instead requires a solution to be found of the lowest degree, both for the FUT and its complement.

The requirement for a lowest degree solution dramatically speeds up the algorithm utilized for this work. Since the AI can be no higher than $\lceil n/2 \rceil$, there is no need to even solve for annihilators of that degree (as it is guaranteed that at least one exists if there is no annihilator of lesser degree). Instead, we search for annihilators of degree one through $\lceil n/2 \rceil - 1$, understanding that if none are found, the AI of the FUT is $\lceil n/2 \rceil$.

An additional speedup is obtained from the knowledge that for all functions other than the two constant functions, the AI must be at least one. Once a degree one annihilator is found, the algorithm stops testing for that function even if the complement has not been checked since the AI can be no lower.

2. Operation for $n = 4$

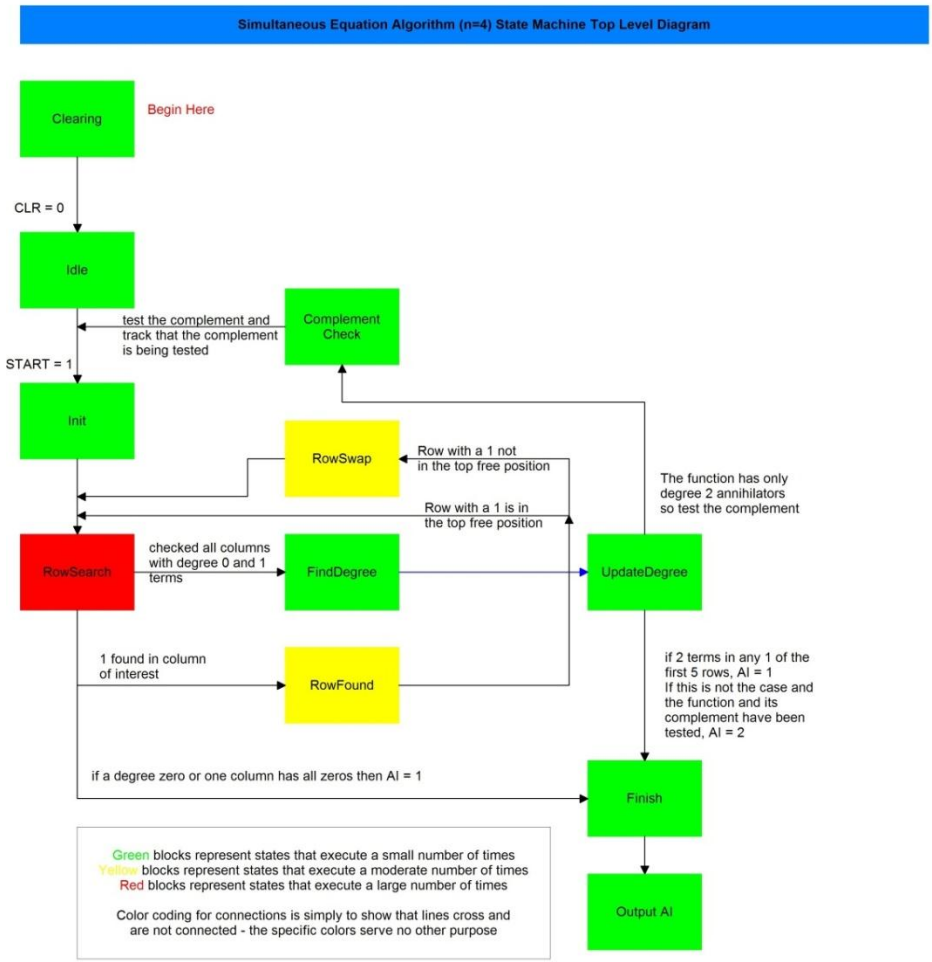

Figure 7. Simultaneous equation algorithm ($n = 4$) state machine top-level diagram.

For $n = 4$, the only possible values for AI are one and two, ignoring the constant functions. The top level diagram governing the operation of the algorithm is shown in Figure 7. After initialization, the algorithm begins searching through the rows, column by column, looking for the leading one in each row, and then it zeros out the remainder of

that column. Rows are swapped if necessary, ultimately resulting in the transformation of the annihilator matrix to reduced row echelon form if the algorithm continues to completion. If the algorithm finds an empty column, the AI is set to one because the empty column represents a free variable, signifying that a degree one solution exists. If the algorithm fails to find an empty column, it searches the nonzero rows for two values in a row for the degree zero and degree one terms. If the algorithm finds two values in a single row, this represents a degree one solution, and the algorithm exits.

If no degree one solutions are found after searching the original TT, the same operation is performed on the complement. Finding a degree one solution, either by finding an empty column or two terms in a row, causes the algorithm to exit and output one for AI. If no degree one solution is found in the complement, the algorithm provides two as the output for AI since there must be a degree two annihilator that is of lowest degree.

3. Operation for $n = 5$

The algorithm for $n = 5$ operates very similar to that for $n = 4$. Rows searches are performed, and the annihilator matrix is placed in reduced row echelon form. Once a degree one annihilator is found, the algorithm exits without continuing its checks. An empty column or two values from the set of degree zero and degree one terms in a single row still represents a degree one solution.

The difference between the algorithms occurs if a degree one solution is not found. This algorithm must search for degree two solutions since a degree three solution is possible for $n = 5$ because $\lceil n/2 \rceil = 3$. The determination of a degree two solution is performed the same as that for degree one. An empty column represents a free variable and, thus, a degree two solution. Two terms from the set of degree zero, one, and two terms in a single row also represents that a degree two solution exists. Once a degree two solution is found, AI is tentatively set to two, and the algorithm tests the complement. If no degree two solution is found, the lowest degree annihilator for the original function is three, and the complement must be tested.

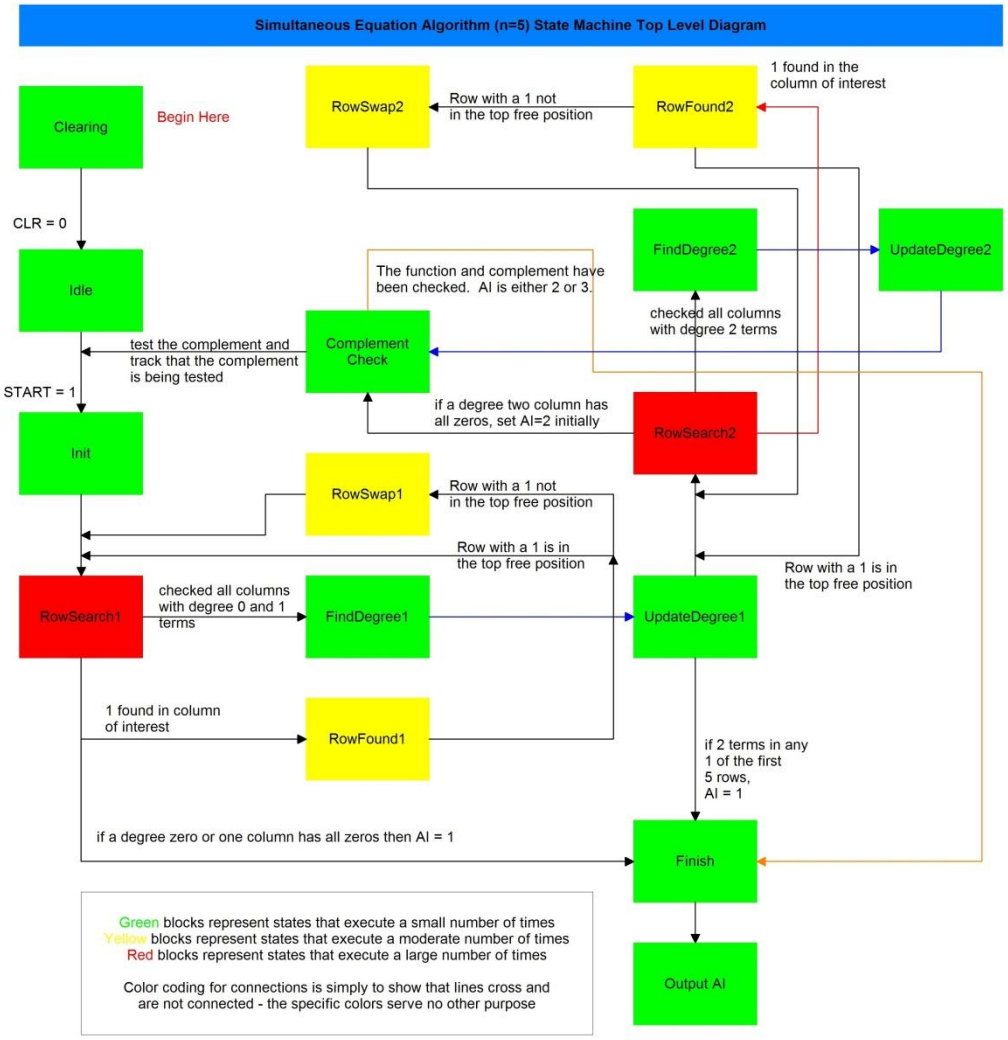

Figure 8. Simultaneous equation algorithm (n = 5) state machine top-level diagram.

If a degree one solution is found for the complement, the algorithm exits with an output of one for AI. If no degree one solution is found for the complement and the original TT had a degree two annihilator, the algorithm exits with an output of two for AI, as shown in Figure 8. Degree two annihilators will only be checked in the complement if no degree one or two annihilators were found in the original TT. If a degree two annihilator is found, the function exits and outputs two for AI. If no degree two annihilator is found, the function exits and outputs three for AI since neither the original function nor its complement had any annihilator lower than degree three.

4. Operation for $n = 6$

Operation for $n = 6$ is the same as for $n = 5$, except that there are more terms in the annihilator matrix. Both numbers of variables have the same maximum for AI, and the same testing is performed for either.

5. Advantages

The primary advantage of this algorithm is that it is significantly faster than the other algorithms. As this algorithm immediately exits at the earliest possible opportunity, unnecessary testing is eliminated. There is no need to look for annihilators of the maximal degree as it known that at least one exists (if no lower degree annihilators exist), and there is no need to continue testing once a degree one solution is found since it is known that no lower degree solution exists except for the constant functions.

Another significant advantage of this algorithm is that with it all annihilators for a given function may be found. As currently implemented, this algorithm only searches for the lowest degree annihilator, but it can easily be altered to provide all annihilators. This is useful when assessing the security of a particular Boolean function. When a function is being attacked, the attackers will utilize as many annihilators as possible in an effort to reduce the system to one as close to linear as possible. The brute force algorithm also finds all annihilators, but many algorithms for computing AI do not find all annihilators.

6. Disadvantages

The disadvantage of this algorithm is that its complexity grows rapidly as n increases. Each time a new highest degree annihilator is permitted, such as when moving from $n = 4$ to 5 or $n = 6$ to 7, another set of states has to be added to the algorithm along with an increasing number of registers and a larger array for the simultaneous equation matrix. Maintaining the optimum efficiency with this algorithm requires a significant amount of coding for each successive value of n.

Each algorithm had strengths and weaknesses. A thorough comparison of the results obtained shows why the simultaneous equation algorithm was the most effective at enumerating algebraic immunity.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. RESULTS

The most significant result of this work is the first known enumeration of algebraic immunity for $n = 5$. This is noteworthy because there are $2^{2^n} = 2^{2^5} = 4{,}294{,}967{,}296$ Boolean functions for $n = 5$, and each function must be tested, along with its complement, to fully enumerate AI. The number of functions with each AI is listed in Table 5 for all Boolean functions through $n = 5$. The newly entered data is in bold.

Table 5. Number of functions with each algebraic immunity through $n = 5$.

	Number of variables (*n*)			
AI	**2**	**3**	**4**	**5**
0	2	2	2	2
1	14	198	10,582	7,666,550
2	0	56	54,952	**4,089,535,624**
3	0	0	0	**197,765,120**
Total	16	256	65,536	4,294,967,296

In the interest of presenting the results in sequence, the results from AI enumeration for $n = 4$ are fully discussed first.

A. FULL ENUMERATION OF ALGEBRAIC IMMUNITY ($N = 4$)

The algorithms that produced the results in this thesis were created initially to test the $n = 4$ case. This case has enough unique functions to require a certain degree of computational effort, but not so many that debugging the algorithm is too difficult. The proper choice for the starting case is critical in initial algorithm development.

1. SRC-6

As previously discussed, two different algorithms were implemented on the SRC-6 to compute AI for $n = 4$. The brute force algorithm was implemented first, and then the simultaneous equation algorithm was developed.

a. Runtime Comparison

The runtime for the brute force algorithm was significantly longer than that for simultaneous equation algorithm. This shows that time spent in algorithm development can yield impressive performance gains. These comparisons are summarized in Table 6.

Table 6. Comparison of brute force and simultaneous equation algorithms ($n = 4$) runtime.

	Brute force	Simultaneous equation
Total clocks	80,748,733	4,946,111
Number of functions	65,536	65,536
Clocks per function	1,232.1	75.47
Total time (sec)	0.807	0.0495
Functions per second	81,160	1,325,000

The simultaneous equation algorithm was able to compute the AI for all functions at $n = 4$ in only 6.13% of the time as the brute force algorithm, representing a 1632% speedup.

b. Resource Utilization Comparison

The two algorithms used a similar amount of the FPGA resources with the simultaneous equation algorithm ultimately showing a slightly improved efficiency, as shown in Table 7.

Table 7. Comparison of brute force and simultaneous equation algorithms (n = 4) resource utilization.

	Brute force	Simultaneous equation	Total
Number of slice flip flops/%	2,931/3%	2,760/3%	88,192/100%
Number of 4 input LUTs/%	2,368/2%	2,343/2%	88,192/100%
Number of occupied slices/%	2,190/4%	2,120/4%	44,096/100%
Total number of 4 input LUTs/%	2,616/2%	2,569/2%	88,192/100%
Frequency	103.4 MHz	109.4 MHz	Not Applicable

Both algorithms use a very small amount of the total available resources. The highest utilization is 4% of occupied slices, which is very minor, so resource utilization is not a concern for either algorithm. The difference in their utilization is also so small as to not be statistically significant.

The most interesting item is the frequency. This denotes the maximum speed at which the designed circuit can operate. As the SRC-6 has a clock frequency of 100 MHz, it is preferable for the circuit frequency to be greater than 100 MHz, although code will still run properly at slightly lower frequencies. The simultaneous equation algorithm is capable of operating 6 MHz faster than the brute force algorithm. This implies that the simultaneous algorithm is more readily scaled to a larger number of variables.

2. C Code

The source code to compute AI for $n = 4$ in C was developed utilizing the Verilog algorithm that first enumerated AI on the SRC-6. The code was compiled using Code::Blocks 10.05 and was executed on a Windows 7 PC with 4 GB of RAM and an Intel® Core™2 Duo P8400 CPU operating at 2.26 GHz. The code is designed for single core operation and does not take advantage of the second core present in the processor.

From Table 8 we see the results of executing the C code for $n = 4$. The enumeration of AI for all functions took a fraction of a second. The total time was obtained by performing 1,000 complete iterations and dividing that time by 1,000.

Table 8. C code runtime ($n = 4$).

	C code
Total time (sec)	0.143006
Number of functions	65,536
Functions per second	458,275

3. SRC-6 and C Code Comparison

The first comparison point for the three methods utilized to enumerate AI for $n = 4$ is compile time. The C code took less than a second to compile, while each version of the SRC-6 code required approximately 5 minutes of compilation time.

The more interesting result is the computation time. The simplest method of comparing the three algorithms was to calculate the number of functions each could process per second. The slowest of the three methods was the brute force algorithm, which was capable of evaluating only 81,160 functions per second. The C algorithm actually outperformed this brute force algorithm, evaluating 458,275 functions per second.

The simultaneous equation algorithm on the SRC-6 was faster than both other methods, evaluating 1,325,000 functions per second. This makes the simultaneous equation algorithm nearly three times faster than the C algorithm and over 16 times faster than the brute force algorithm.

B. FULL ENUMERATION OF ALGEBRAIC IMMUNITY ($N = 5$)

The primary goal of this work was to complete the first ever enumeration of AI for all Boolean functions for $n = 5$. That goal was first completed utilizing the simultaneous equation algorithm on the SRC-6. The brute force algorithm proved too slow to enumerate all functions for $n = 5$; although, it was compiled and tested on a small number of functions.

It was believed that a C algorithm would prove too slow to enumerate AI for all functions at $n = 5$, but the algorithm was developed for potential use in some Monte Carlo trials. After development and initial testing, it was discovered that the C algorithm operated quickly enough that full enumeration could be performed for $n = 5$.

One difficulty with computing the distribution of AI among functions for $n = 5$ for the first time is determining if the computation is accurate. Fortunately, there is an existing theoretical calculation that has proven the number of functions which have an AI of one for any number of variables [17]. The calculated value matches precisely with the computational results, providing confidence that the determination of AI for $n = 5$ was performed correctly.

1. SRC-6

When the brute force algorithm was extended to the $n = 5$ case, it was quickly discovered that it operated too slowly to enumerate AI for all functions. This is because for unbalanced functions, the counter in the annihilator generator may have to count as high as 2^{2^n-1}, which is 2^{31} for $n = 5$. This caused the computation of AI for some individual functions to take nearly a minute, precluding the possibility of computing AI for all 2^{32} functions.

a. Runtime Comparison

The runtime for the brute force algorithm was computed based on only ten trials. This is because the algorithm takes so long to compute AI for unbalanced functions that it is not effective to use it for $n = 5$. The runtime for the simultaneous equation algorithm is based on complete enumeration of AI for $n = 5$. These comparisons are summarized in Table 9.

Table 9. Comparison of brute force and simultaneous equation algorithms ($n = 5$) runtime.

	Brute Force	Simultaneous Equation
Total clocks	13,421,773,163	1,496,439,942,292
Number of functions	10	4,294,967,296
Clocks per function	1,342,177,316.3	348.4
Total Time (sec)	134.2	14964.4
Functions per second	0.0745	287012.3

The data for the brute force algorithm represents its worst case scenario for calculating AI for functions. Given this worst case performance, the simultaneous equation algorithm was nearly four million times faster. For the best case scenario, i.e., all perfectly balanced functions, the brute force algorithm would still need to count to 2^{16}, so each function would require a minimum of 65,536 clocks. The average simultaneous equation algorithm performance is 188 times faster than this best case performance.

b. Resource Utilization Comparison

The difference in the algorithms is also apparent when looking at the resource utilization data in Table 10. Here we see that the simultaneous equation algorithm is using more resources than the brute force algorithm. This is because the

complexity of the simultaneous equation algorithm increases more rapidly than for the brute force algorithm, due primarily to the increased number of registers required for all of the arrays.

Table 10. Comparison of brute force and simultaneous equation algorithms (n = 5) resource utilization.

	Brute Force	Simultaneous Equation	Total
Number of slice flip flops/%	3,278/3%	4,110/4%	88,192/100%
Number of 4 input LUTs/%	3,811/4%	5,037/5%	88,192/100%
Number of occupied slices/%	2,987/6%	3,780/8%	44,096/100%
Total number of 4 input LUTs/%	4,091/4%	5,471/6%	88,192/100%
Frequency	63.3 MHz	100.9 MHz	Not Applicable

The most significant difference between the two algorithms is the frequency. For the simultaneous equation algorithm, projections show that it will continue to operate above 100 MHz, the actual frequency of the SRC-6. The brute force algorithm slows down to 63.3 MHz. At that frequency, operation is less reliable, and the results can no longer be guaranteed to be accurate. This is another reason for performing so few trials with the brute force algorithm. This result also correlates with the expectation that the simultaneous equation algorithm would operate properly for higher values of n.

2. C Code

The source code to compute AI for $n = 5$ in C is an extension of the code used to compute AI for $n = 4$. This code continues to closely follow the algorithm that is utilized for computing AI using the SRC-6.

In Table 11, we see the results of executing the C code for $n = 5$. The enumeration of AI for all functions with this number of variables took significantly longer than for the $n = 4$ case.

Table 11. C code runtime ($n = 5$).

	C code
Total time (sec)	72760.026
Number of functions	4,294,967,296
Functions per second	59029.2

The C code required nearly eight times as much processing time per function as compared to the $n = 4$ case.

3. SRC-6 and C Code Comparison

We will again start the comparison with compile time. The C code for $n = 5$ required less than a second to compile, while each version of the SRC-6 code required approximately 15 minutes of compilation time. The time difference for creating the C executable was not noticeable between this case and the $n = 4$ case. The SRC-6 code required three times as much compile time as the previous case and significantly more time than the C code.

Comparing the number of functions processed per second, we see a more substantial difference for the $n = 5$ case as compared to $n = 4$. The brute force method was again the slowest of the three, a condition exacerbated by only testing it for worst-

case functions. It processed only 0.0745 functions per second, making it orders of magnitude slower than either of the other functions.

The real comparison for $n = 5$ is between the simultaneous equation algorithm on the SRC-6 and the same algorithm implemented in C. The SRC-6 continued to outperform the C code, processing 287012.3 functions per second. The C code processed only 59029.2 functions per second.

The SRC-6 computed AI for functions at a rate 4.86 times faster than the equivalent C code. This is an improvement over the $n = 4$ case when it was only approximately three times faster than the equivalent C code.

Much of this speedup is due to the manner in which the SRC-6 processes matrices. A conventional processor has to manipulate variables in a matrix one operation at a time, and if the matrix is large enough multiple operations may be required to manipulate a single element. The SRC-6 can handle multiple operations simultaneously, and this is a powerful capability for higher degree functions. When putting a matrix into reduced row echelon form, the entire matrix can be updated in a single clock period each time the leading coefficient is found in the column being process.

C. PARTIAL ENUMERATION OF ALGEBRAIC IMMUNITY ($N = 6$)

There are 2^{64} unique functions for $n = 6$. This is far too many functions to compute AI for each individual function with current technology in a reasonable amount of time. In order to extend AI determination to $n = 6$, Monte Carlo techniques were employed.

For Monte Carlo trials, a large number of random functions are tested to generate a distribution of expected values for the property being tested. In this case, large numbers of functions were tested to determine their AI using these random trials.

In order for the distribution produced by Monte Carlo techniques to be accurate, the numbers used for the random trials must either be truly random or be pseudorandom numbers with appropriate statistical properties. For this work, a version of the Mersenne Twister algorithm was used to generate pseudorandom numbers. The Mersenne Twister

algorithm has excellent statistical properties which make it well suited to produce pseudorandom numbers for Monte Carlo trials [18]. The algorithm was seeded with a truly random number obtained from random.org. The same seed was used for both the C code and the SRC-6 code so that the same pseudorandom sequence would be utilized in each to aid in comparison.

Completing 500 million iterations provided sufficient data to form an estimate of the distribution of algebraic immunity among functions for $n = 6$, and this is shown in Table 12. The estimated number of functions with an AI of one is 1,143,698,132,570. This matches closely with the calculated value of 1,081,682,871,734 [17].

Table 12. Number of functions with each algebraic immunity through $n = 6$.

	Number of variables (*n*)				
AI	**2**	**3**	**4**	**5**	**6**
0	2	2	2	2	2
1	14	198	10,582	7,666,550	1,081,682,871,734
2	0	56	54,952	**4,089,535,624**	***1,269,431,213,963,372,798***
3	0	0	0	**197,765,120**	***17,177,311,716,048,046,248***
Total	16	256	65,536	4,294,967,296	18,446,744,073,709,551,616

We can see the estimated distribution of AI among functions for $n = 6$ in Table 12, where the estimated numbers are italicized. The numbers for AI of 0 and 1 are known, as are the total number of functions. The estimates for an AI of 2 and 3 are based on 500 million iterations of the C algorithm. This represents a sample size of $2.7105 \cdot 10^{-9}\%$. You can see the exact number of functions with each AI in Table 13.

Table 13. Result of 500 million AI computations for $n = 6$ in C.

Algebraic Immunity	Number of Functions
1	31
2	34,408,002
3	465,591,967

1. SRC-6

The extension of the simultaneous equation algorithm to $n = 6$ involved a relatively simple modification of the code. This is primarily because for $n = 6$ there are the same possible values for AI as for $n = 5$, so no entirely new portions of code had to be designed. An extension to $n = 7$ would not be as simple.

The difficulty with performing random trials came from the generation of pseudorandom numbers. There are built-in macros that can aid in producing pseudorandom numbers, but they require that the user macro be pipelined. The simultaneous equation algorithm is state machine based, preventing the use of built-in randomization. Some attempts were made to implement random functions in Verilog, but these were unsuccessful. With each compile requiring more than 24 hours, there was insufficient time to continue further experimentation with Verilog randomization.

Instead, the same Mersenne Twister code that was used in the C algorithm was implemented in the main.c file, allowing that file to pass random numbers to the FPGA for processing. This introduces a significant delay as compared to generating random numbers in Verilog, and the SRC-6 documentation does not adequately quantify this delay. This prevents a meaningful comparison of the SRC-6 implementation and the C implementation for $n = 6$. The resource utilization data is provided in Table 14.

Table 14. Simultaneous equation algorithm resource utilization on the SRC-6 (n = 6).

	Simultaneous Equation	Total
Number of Slice Flip Flops/%	3,235/3%	88,192/100%
Number of 4 input LUTs/%	8,990/10%	88,192/100%
Number of occupied Slices/%	5,060/11%	44,096/100%
Total Number of 4 input LUTs/%	9,010/10%	88,192/100%
Frequency	87.5 MHz	Not Applicable

The large number of arrays required for n = 6 only requires 11% of the resources of a single FPGA. The frequency has dropped to only 87.5 MHz, indicating that the circuit is not meeting all time constraints.

2. C Code

The C code for n = 6 was again a simple extension from the code used for n = 5. The code required only a few seconds of time to compile. The code ran for 500 million iterations to provide the distribution of AI for n = 6, and its performance characteristics can be seen in Table 15.

Table 15. C code runtime ($n = 6$).

	C code
Total time (sec)	26673.582
Number of functions	500,000,000
Functions per second	18745.14

The C code computed AI for $n = 6$ at approximately one-third of the rate for $n = 5$. A significant contributor to this slowdown is the distribution of functions with an AI of three for $n = 6$. Those functions take the most time to process because they require the code to execute in its entirety.

3. SRC-6 and C Code Comparison

Randomization for the SRC-6 was implemented by using the Mersenne Twister algorithm in main.c to send individual random functions to the macro for testing. This unnecessarily slowed the execution time of the SRC-6 randomization and makes the comparison between it and the C code inaccurate, but the evaluation provides some discussion points. The runtimes for each can be seen in Table 16.

Table 16. SRC-6 and C code runtime comparison ($n = 6$).

	C code	SRC-6 simultaneous equation
Total time (sec)	26673.582	1949.51
Number of functions	500,000,000	25,000,000
Functions per second	18745.14	12823.74

The C code performed 46% faster than the SRC-6 simultaneous equation algorithm for $n = 6$. This shows the viability of the SRC-6 for computing AI with larger numbers of variables. Hindered by slow memory transfers for every computation, the SRC-6 remained nearly at parity in performance with the C code algorithm. A Verilog-based pseudorandom number generator would provide a better measure of the SRC-6's performance for $n = 6$.

These results demonstrate the power of the SRC-6 in computing algebraic immunity. The most complex algorithm used for computing AI required a small portion of the total resources of the SRC-6. Tremendous potential exists to expand this work to larger numbers of variables.

VII. CONCLUSION AND RECOMMENDATIONS

A. CONCLUSION

The first known computation of algebraic immunity for all Boolean functions with five variables was successfully completed. This computation was carried out both on the SRC-6 using a simultaneous equation algorithm and on a conventional processor using an algorithm developed in C. The results obtained through these computations matched each other. The computed number of functions with an AI of one exactly matched the calculated value. The calculated value is a proven number, which validates the results obtained.

Monte Carlo trials were performed to estimate the number of functions with each algebraic immunity for six variable Boolean functions. The number of estimated functions with an AI of one deviated from the calculated value by less than 6 percent.

The first known Verilog implementation of a reduced transeunt triangle was utilized for the brute force algorithm for computing algebraic immunity. This reduced transeunt triangle has n delay versus 2^n delay for a complete transeunt triangle and requires significantly fewer gates. Its Verilog design is recursive in nature, allowing for easy expansion to accept arbitrary numbers of inputs.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

1. Monte Carlo Trials for $n = 6$ Using Verilog Randomization

The next significant step in expanding this work is to provide an accurate assessment of its runtime in comparison to C code for $n = 6$. To accomplish this, a pseudorandom number generator in Verilog must be implemented to eliminate the delay created by passing all parameters via memory. There are available Verilog implementations that should be adaptable for this purpose.

2. Monte Carlo Trials for $n = 7$ and $n = 8$

The next logical extension of the work is to perform random trials for $n = 7$ and $n = 8$. Testing these functions will provide a better estimate of the distribution of AI at

higher functions and bring this work closer to the number of variables utilized in actual cryptosystems. This will require the computation of degree three annihilators, since functions with these numbers of variables can have AI up to four.

3. Nonlinearity Sieve

A correlation exists between nonlinearity and algebraic immunity. This relationship can be exploited to speed up the process of calculating AI or to more quickly search for functions with a specific AI. Previous thesis work at the Naval Postgraduate School has produced an algorithm that determines nonlinearity in a single clock on the SRC-6. This work could be used to efficiently implement a nonlinearity sieve.

4. Equivalence Classes

There are equivalence classes where each function in the class has the same AI. The number of such classes is known, and there are listings containing a representative from each class. These classes could be used to determine the complete distribution of AI for numbers of variables where it is computationally infeasible to test each individual function.

5. Algorithm Modularity

The code utilized for AI computation is efficient, but code creation is difficult and time consuming. Making the code modular would allow it to be simply expanded to larger numbers of variables so that specific classes of functions could be tested to determine their algebraic immunity.

APPENDIX A. SRC-6 SOURCE CODE

The source code for the SRC-6 is divided into six separate files: Makefile, main.c, subr.mc, macro.v, info, and blk.v. The macro.v, info, and blk.v files are only required if a user macro is being implement; although, the power of the SRC-6 is its ability to use custom macros to perform a desired computation. The first three files reside in the working directory, and the final three files are typically placed in a folder called "my_macro" inside of the working directory. The locations of files can be changed if desired so long as the Makefile is appropriately edited.

For this thesis, the directory structure discussed is utilized. The macro file is named Algebraic_Immunity.v instead of macro.v. For each case whose files are provided, all files are included to aid those desiring to repeat this work.

All source code was formatted with Notepad++.

A.1 COMMON SRC-6 FILES

The Makefile, blk.v, and info files did not change between cases, so one set of these is included in the initial section. The main.c, subr.mc, and Algebraic_Immunity.v files did change, and the new files are included for each case.

1. Makefile

```
# $Id: Makefile.template,v 1.13 2005/04/12 19:18:30 jls Exp $
#
# Copyright 2003 SRC Computers, Inc.  All Rights Reserved.
#
#       Manufactured in the United States of America.
#
# SRC Computers, Inc.
# 4240 N Nevada Avenue
# Colorado Springs, CO 80907
# (v) (719) 262-0213
# (f) (719) 262-0223
#
# No permission has been granted to distribute this software
# without the express permission of SRC Computers, Inc.
#
# This program is distributed WITHOUT ANY WARRANTY OF ANY KIND.
#
# ---------------------------------
```

```
# ----------------------------------
# User defines FILES, MAPFILES, and BIN here
# ----------------------------------
FILES           = main.c

MAPFILES        = subr.mc

BIN             = main

# ----------------------------------
# Multi chip info provided here
# (Leave commented out if not used)
# ----------------------------------
#PRIMARY        = <primary file 1>   <primary file 2>

#SECONDARY      = <secondary file 1> <secondary file 2>

#CHIP2          = <file to compile to user chip 2>

#----------------------------------
# User defined directory of code routines
# that are to be inlined
#----------------------------------

#INLINEDIR      =


# ----------------------------------
# User defined macros info supplied here
#
# (Leave commented out if not used)
# ----------------------------------
MACROS          = my_macro/Algebraic_Immunity.v
MY_BLKBOX       = my_macro/blk.v
MY_NGO_DIR      = my_macro
MY_INFO         = my_macro/info
# ----------------------------------
# Floating point macros selection
# ----------------------------------

#FPMODE         = SRC_IEEE_V1 # Default SRC version IEEE
#FPMODE         = SRC_IEEE_V2 # Size reduced SRC IEEE with
                               # special rounding mode
# ----------------------------------
# User supplied MCC and MFTN flags
# ----------------------------------

MCCFLAGS     = -v
MFTNFLAGS    = -v

# ----------------------------------
# User supplied flags for C & Fortran compilers
# ----------------------------------
```

```
CC            = gcc   # icc   for Intel cc for Gnu
FC            = ifort # ifort for Intel f77 for Gnu
#LD           = ifort -nofor_main # for mixed C & Fortran, main in C
#LD           = ifort # for Fortran or C/Fortran mixed, main in Fortran
LD            = gcc   # for C codes

MY_CFLAGS     =
MY_FFLAGS     =
MY_LDFLAGS    =      # Flags to include libs if needed
# ----------------------------------
# VCS simulation settings
# (Set as needed, otherwise just leave commented out)
# ----------------------------------

#USEVCS         = yes  # YES or yes to use vcs instead of vcsi
#VCSDUMP        = yes  # YES or yes to generate vcd+ trace dump
# ----------------------------------
# MODELSIM simulation settings
# (Set as needed, otherwise just leave commented out)
# ----------------------------------

#USEMDL         = yes  # YES or yes to use modelsim instead of vcs/vcsi
#USEMDLGUI      = yes  # YES or yes to use modelsim GUI interface
#MDLDUMP        = yes  # YES or yes to generate vcd trace dump
# ----------------------------------
# No modifications are required below
# ----------------------------------
MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)
```

2. info.v

```
//*****************************************************************
//
//  info - info file to specify the input and output of the macro ...
//
//       Author:         Eric McCay
//       Created:        July 25, 2011
//
//*****************************************************************

BEGIN_DEF "my_operator"         //Name used in .mc file to call macro.
        MACRO = "Algebraic_Immunity";       //Macro name.
        STATEFUL = YES;
        EXTERNAL = YES;
        PIPELINED = NO;
        LATENCY = 0;

        INPUTS = 2:
            I0 = INT 64 BITS (TT[63:0]) //TT of function under test
            I1 = INT 1 BITS (START)     //For initialization
            ;
```

```
        OUTPUTS = 2:
            O0 = INT 64 BITS (AI[63:0]) // Output Algebraic Immunity
            O1 = INT 1 BITS (DONE)      // Indicates completion
            ;

        IN_SIGNAL: 1 BITS "CLK" = "CLOCK";  //Clock input
        IN_SIGNAL: 1 BITS "CLR" = "code_block_reset";

END_DEF
```

3. blk.v

```
//*************************************************************
//
//  blk.v - A blackbox file that specifies inputs and outputs
//
//       Author:         Eric McCay & Jon T. Butler
//       Created:        July 25, 2011
//
//*************************************************************

module Algebraic_Immunity (TT, AI, DONE, CLK, CLR, START);
    input [63:0] TT;
    output [63:0] AI;
    output DONE;
    input CLK;
    input CLR;
    input START;
endmodule
```

A.2 BRUTE FORCE STATE MACHINE ALGORITH (*n* = 4)

The original brute force algorithm for the SRC-6 functioned properly for the $n = 4$ case but is too slow to enumerate all functions for $n = 5$.

1. main.c

```
//*********************************************************************
//
//  main.c  -    C program to run Algebraic_Immunity
//
//       Author:         Eric McCay
//       Created:        July 25, 2011
//
//       Description:  This program determines the Algebraic Immunity
//                      of all Boolean functions for a given n and
//                      provides an output specifying the number of
//                      functions with each AI.
```

```
//
//
//*********************************************************************

#include <map.h>
#include <stdlib.h>
#include <stdio.h>

void subr (int64_t*, int64_t*, int ); //declaration for subr.mc

int main (int argc, char *argv[]) {
    FILE *res_map, *res_cpu;
        int mapnum = 0; //specify which map is used
        int i;
        int64_t time_clock; //used to track runtime
        int64_t *AI;

// Allocate array of for the return of AI values
    AI = (int64_t *) malloc (4* sizeof (int64_t));

// Set TT to all possible values

        for (i = 0; i < 4; i++){
            AI[i] = 0;              //Zero out AI.
            }

    map_allocate (1);   // reserves map 1

    //This shows that the subr.mc has been called.  Subroutine
    //calls can take a considerable amount of time so this lets
    //the user know that execution has started properly.
    printf ("Calling subr.mc\n\n");

//  Call subroutine subr.mc on the MAP.
    subr (AI, &time_clock, mapnum);

    printf("Return from subr.mc\n\n");

//  Print out the number of clocks.
printf ("%lld clocks\n", time_clock);

/*  Print out the Algebraic Immunity of each Function   */
    printf("Listed below is the number of functions with each "
            "Algebraic Immunity\n\n");


        printf("AI = 3: %d\n",AI[3]);
        printf("AI = 2: %d\n",AI[2]);
        printf("AI = 1: %d\n",AI[1]);
        printf("AI = 0: %d\n",AI[0]);

    map_free (1); // release the map we were using

    exit(0);
```

```
}//int main (int argc, char *argv[]) {
```

2. subr.mc

```
//*****************************************************************
//
//  subr.mc  - MAP C subroutine to determine Algebraic Immunity
//
//       Author:         Eric McCay
//       Created:        July 25, 2011
//
//       Description:  This program calls Algebraic_Immunity.v, which
//                     determines the Algebraic Immunity of the
//                     function provided in Truth Table Form.
//
//*****************************************************************

#include <libmap.h>
#define NUM 65536       //number of values in TT 2^(2^n)

void subr (int64_t ai[], int64_t *time, int mapnum) {

// Declare one OBM bank in the SRC-6 to store the number of
// functions with each possible AI value.
        OBM_BANK_B (AI, int64_t, 4)

        int64_t t0, t1; // Used to determine runtime
        int64_t my64bit_in; //input TT to test
        int64_t my64bit_out; //output AI of tested function
        int i, j, k, l, m, n;

        read_timer(&t0);

        for (i = 0; i < 4; i++)
            AI[i] = 0; //Initially, zero out the AI values
        k = 0;
        l = 0;
        m = 0;
        n = 0;

        //This for loop calls the macro file the required number
        //of times (65536 in this case) to determine the AI
        //for each possible TT input on 4 variables.  It then
        //uses a switch statement to tally the results for
        //each possible AI value.  For n=4, AI can be at most
        //two, so the case 3 statement never executes.
        for (i = 0; i < NUM; i++)
            {
                 my64bit_in = i;
                 my_operator (my64bit_in, &my64bit_out);
                 j = my64bit_out;
                 switch (j)
                 {
```

```
                case 0:
                    k++;
                    break;
                case 1:
                    l++;
                    break;
                case 2:
                    m++;
                    break;
                case 3:
                    n++;
                    break;
                }

            }//for (i = 0; i < NUM; i++){

        AI[0] = k;
        AI[1] = l;
        AI[2] = m;
        AI[3] = n;

        read_timer(&t1);

        *time = (t1 - t0);

// Return AI values by DMAing TO the CPU
        DMA_CPU (OBM2CM, AI, MAP_OBM_stripe(1,"B"), ai,
                 1, 4*sizeof(int64_t), 0);
        wait_DMA (0);

}
```

3. Algebraic_Immunity.v

```
module Ones_Count (TT_ext, Count);
//-----------------------------------------------------------------------
// Ones_Count.v -  A program to count the 1's in a variety of inputs,
//                 from 2 - 8 variables
//
// Created:       August 18, 2007
// Author:        Jon T. Butler
// Modified by:   Eric McCay
//
// Inputs:        TT_ext  2-8-variable Truth Table
// Outputs:       Count Number of 1's
//
// Notes:        1.  parameter n is used to specify that a n-variable
//                   function's truth table is being considered
//                   (TT has 2^n-inputs).
//-----------------------------------------------------------------------

        parameter n = 2;
        localparam N = 2**n;
```

```
        input[N-1:0] TT_ext;    //Function Truth Table
        output[N:0] Count;  //Need 9 bits to represent all possible
                            //counts for 8 variables
        reg[N:0] Count;
        wire[N-1:0] TT;

        generate
        assign TT = TT_ext;
        endgenerate

        always @(TT)
           begin: CHECK_n
               case(n) // Call appropriate case for the size of n
                   2: Count = Count2(TT);
                   3: Count = Count3(TT);
                   4: Count = Count4(TT);
                   5: Count = Count5(TT);
                   6: Count = Count6(TT);
                   7: Count = Count7(TT);
                   8: Count = Count8(TT);
                   default Count = Count2(TT);
               endcase
           end

//---------------------------------------------------------------------
//------  The 1's count function - Count2 for 2-variable functions   --
function [8:0] Count2;
   input [3:0] TT;
   begin: f2
      Count2[0]=TT[3]^TT[2]^TT[1]^TT[0];
      Count2[1]=(TT[3]&TT[2]|TT[3]&TT[1]|TT[3]&TT[0]|TT[2]&TT[1]|TT[2]
                &TT[0]|TT[1]&TT[0])&~(TT[3]&TT[2]&TT[1]&TT[0]);
      Count2[2]=TT[3]&TT[2]&TT[1]&TT[0];
      Count2[8:3]=6'b000000;
   end
endfunction

//------  The 1's count function - Count2 for 2-variable functions   --
//---------------------------------------------------------------------

// For n = 3 and on, it just recursively calls the previous count
// function. So, for example, for n = 4, count4 is called, which calls
// count3 twice, which calls count2 a total of 4 times, and this does
// the appropriate amount of counting.
//---------------------------------------------------------------------
//------  The 1's count function - Count3 for 3-variable functions   --
function [8:0] Count3;
   input [7:0] TT;
   begin: f3
      Count3 = Count2(TT[7:4]) + Count2(TT[3:0]);
   end
endfunction

//------  The 1's count function - Count3 for 3-variable functions   --
```

```
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
//------  The 1's count function - Count4 for 4-variable functions   --
function [8:0] Count4;
   input [15:0] TT;
   begin: f4
      Count4 = Count3(TT[15:8]) + Count3(TT[7:0]);
   end
endfunction

//------  The 1's count function - Count4 for 4-variable functions   --
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
//------  The 1's count function - Count5 for 5-variable functions   --
function [8:0] Count5;
   input [31:0] TT;
   begin: f5
      Count5 = Count4(TT[31:16]) + Count4(TT[15:0]);
   end
endfunction

//------  The 1's count function - Count5 for 5-variable functions   --
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
//------  The 1's count function - Count6 for 6-variable functions   --
function [8:0] Count6;
   input [63:0] TT;
   begin: f6
      Count6 = Count5(TT[63:32]) + Count5(TT[31:0]);
   end
endfunction

//------  The 1's count function - Count6 for 6-variable functions   --
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
//------  The 1's count function - Count7 for 7-variable functions   --
function [8:0] Count7;
   input [127:0] TT;
   begin: f7
      Count7 = Count6(TT[127:64]) + Count6(TT[63:0]);
   end
endfunction

//------  The 1's count function - Count7 for 7-variable functions   --
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
//------  The 1's count function - Count8 for 8-variable functions   --
function [8:0] Count8;
   input [255:0] TT;
   begin: f8
```

```
      Count8 = Count7(TT[255:128]) + Count7(TT[127:0]);
   end
endfunction

//------  The 1's count function - Count8 for 8-variable functions   --
//-----------------------------------------------------------------------
endmodule

module ANF_to_Degree(ANF, degree);
//-----------------------------------------------------------------------
// ANF_to_Degree -  Verilog code to produce, Degree, the highest degree
//                  of the ANF of an input function.
//
// Created:       July 28, 2011
// Author:        Eric McCay and Jon T. Butler
//
// Inputs:        ANF    - Binary 2^n-tuple ANF of given function
// Outputs:       Degree - Highest degree of ANF that is less than
//                         parameter ignore.
//
//-----------------------------------------------------------------------
//

parameter n = 2;                // The number of variables.
localparam N = 2**n;            // Max number of elements in ANF.
localparam n_degr = clogb2(n); // The number of bits needed to
                                // represent n, the largest
                                // possible degree.

input  [N-1:0]      ANF;
output [n_degr-1:0] degree;
reg    [n_degr-1:0] degree;
reg    [n:0]        deg;    // deg[i] = 1 iff there is at least one
                            // term in the ANF of degree i.
integer i;

always @(ANF)
    begin
        deg = {(n+1){1'b0}};
        //synthesis loop_limit 32000
        for (i = 0; i < N; i = i + 1)
            begin
                if((ones_counter(i) <= n) && (ANF[i] == 1'b1))
                    deg[ones_counter(i)] = 1'b1;
            end
    end

always @(deg)
    begin
        degree = 0;
            for (i = 0; i <= n; i = i+1)
                if(deg[i] == 1'b1)
                    degree = i;
    end
```

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

//Constant function to produce nbits_fact_n(n)
function integer ones_counter(input integer n);
integer m;
    begin
        ones_counter = 0;
        m = n;
        while (m > 0)
            begin
                ones_counter = ones_counter + m%2;
                m = m >> 1;
            end
    end
endfunction

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//Constant function
function integer clogb2(input integer depth);
    begin
        for(clogb2=0; depth>0; clogb2 = clogb2 + 1)
            depth = depth >> 1;
    end
endfunction

endmodule

module TranseuntTriangleToDegree(TT_ext, Deg_out);
//----------------------------------------------------------------------
// Transeunt_Triangle -  A module to convert between ANF and TT form
//
// Created:       January 21, 2012
// Author:        Eric McCay
//
// Inputs:        TT_ext  - input in TT or ANF form
// Outputs:       TT_out  - output in ANF form or TT form
//
// Notes:     1.  parameter n is used to specify that a n-variable
//                function's truth table is being considered
//                (TT has 2^n-inputs).
//----------------------------------------------------------------------

    parameter n = 3;
    localparam N = 2**n;
    localparam n_degr = clogb2(n); // The number of bits needed to
                                   // represent n, the largest possible
                                   // degree.

    input   [N-1:0]         TT_ext;     // Function Truth Table
    reg     [N-1:0]         Alt_form;   // Stores converted ANF or TT,
                                        // depending on input
    wire    [N-1:0]         TT_in;
    output  [n_degr-1:0]    Deg_out;
```

```
    defparam U1.n = n; // maintains code parameterization

    ANF_to_Degree U1 (Alt_form, Deg_out); // convert the computed ANF
                                          // to its degree

    generate
        assign TT_in = TT_ext;
    endgenerate

    always @(TT_in)
        begin: CHECK_n
            case(n)       //  Call appropriate case for the size of n
                2: Alt_form = TransTri2(TT_in);
                3: Alt_form = TransTri3(TT_in);
                4: Alt_form = TransTri4(TT_in);
                5: Alt_form = TransTri5(TT_in);
                6: Alt_form = TransTri6(TT_in);
                7: Alt_form = TransTri7(TT_in);
                8: Alt_form = TransTri8(TT_in);
                default Alt_form = TransTri2(TT_in);
            endcase

        end

//-------------------------------------------------------------------------
//-The Transeunt Triangle function - TransTri2 for 2-variable functions
function [3:0] TransTri2;
   input [3:0] TT_in;
   begin: f2
      TransTri2[0]=TT_in[0];
      TransTri2[1]=TT_in[0]^TT_in[1];
      TransTri2[2]=TT_in[0]^TT_in[2];
      TransTri2[3]=(TT_in[0]^TT_in[1])^(TT_in[2]^TT_in[3]);
   end
endfunction

//-The Transeunt Triangle function - TransTri2 for 2-variable functions
//-------------------------------------------------------------------------

// For n = 3 and on, it just recursively calls the previous TransTri
// function. So, for example, for n = 4, TransTri4 is called, which
// calls TransTri3 twice, which calls TransTri2 a total of 4 times.
//-------------------------------------------------------------------------
//-The Transeunt Triangle function - TransTri3 for 3-variable functions
function [7:0] TransTri3;
   input [7:0] TT_in;
   begin: f3
       TransTri3[3:0] = TransTri2(TT_in[3:0]);
       TransTri3[7:4] = TransTri2(TT_in[7:4])^TransTri3[3:0];
   end
endfunction

//-------------------------------------------------------------------------
//-The Transeunt Triangle function - TransTri4 for 4-variable functions
```

```
function [15:0] TransTri4;
   input [15:0] TT_in;
   begin: f4
       TransTri4[7:0] = TransTri3(TT_in[7:0]);
       TransTri4[15:8] = TransTri3(TT_in[15:8])^TransTri4[7:0];
   end
endfunction

//-----------------------------------------------------------------------
//-The Transeunt Triangle function - TransTri5 for 5-variable functions
function [31:0] TransTri5;
   input [31:0] TT_in;
   begin: f5
       TransTri5[15:0] = TransTri4(TT_in[15:0]);
       TransTri5[31:16] = TransTri4(TT_in[31:16])^TransTri5[15:0];
   end
endfunction

//-----------------------------------------------------------------------
//-The Transeunt Triangle function - TransTri6 for 6-variable functions
function [63:0] TransTri6;
   input [63:0] TT_in;
   begin: f6
       TransTri6[31:0] = TransTri5(TT_in[31:0]);
       TransTri6[63:32] = TransTri5(TT_in[63:32])^TransTri6[31:0];
   end
endfunction

//-----------------------------------------------------------------------
//-The Transeunt Triangle function - TransTri7 for 7-variable functions
function [127:0] TransTri7;
   input [127:0] TT_in;
   begin: f7
       TransTri7[63:0] = TransTri6(TT_in[63:0]);
       TransTri7[127:64] = TransTri6(TT_in[127:64])^TransTri7[63:0];
   end
endfunction

//-----------------------------------------------------------------------
//-The Transeunt Triangle function - TransTri8 for 8-variable functions
function [255:0] TransTri8;
   input [255:0] TT_in;
   begin: f8
       TransTri8[127:0] = TransTri7(TT_in[127:0]);
       TransTri8[255:128] = TransTri7(TT_in[255:128])^TransTri8[127:0];
   end
endfunction

//Constant function to produce nbits_fact_n(n)
function integer ones_counter(input integer n);
integer m;
    begin
        ones_counter = 0;
        m = n;
        while (m > 0)
```

```
            begin
                ones_counter = ones_counter + m%2;
                m = m >> 1;
            end
    end
endfunction

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
//Constant function
function integer clogb2(input integer depth);
    begin
        for(clogb2=0; depth>0; clogb2 = clogb2 + 1)
            depth = depth >> 1;
    end
endfunction

endmodule

module Algebraic_Immunity(TT, AI, DONE, CLK, CLR, START);
//----------------------------------------------------------------------
// Algebraic_Immunity -  Verilog code to determine the algebraic
//                       immunity of the provided function.
//
// Created:       August 24, 2011
// Author:        Eric McCay and Jon T. Butler
//
// Inputs:        TT - Truth table of the function being tested
// Outputs:       AlgebraicImmunity - The algebraic immunity of the
//                                    tested function
//
//----------------------------------------------------------------------
//

parameter n = 4;                // The number of variables.
localparam N = 2**n;            // Max number of elements in ANF.
localparam n_degr = clogb2(n);  // The number of bits needed to
                                // represent n, the largest possible
                                // degree

input           CLK;
input           CLR;
input           START;
input   [63:0]  TT; // The function under test
wire    [63:0]  TT;
reg     [63:0]  TT_reg;
reg     [63:0]  GlobalMinimum; // Used to store the minimum annihilator
output  [63:0]  AI;     // Algebraic Immunity of function under test
output          DONE;   // Indicates completion
reg             DONE;
reg     [63:0]  AI;
reg     [63:0]  Max; //The maximum possible value of Algebraic Immunity

reg     [N-1:0] Counter; // Used to cycle through possible annihilators
```

```
wire [n_degr-1:0]    f_degree_wire;
wire [n_degr-1:0]    f_bar_degree_wire;

reg     [N-1:0]      TT_annihilator_fbar; // Annihilator function for f
reg     [N-1:0]      TT_annihilator_f;
reg                  TT_annihilator_fbar_inhibit;
reg                  TT_annihilator_f_inhibit;
reg     [N-1:0]      number_ones_fbar [0:N]; // Used to track which
                     //counter an output of annihilator_fbar connects to
reg     [N-1:0]      number_ones_f [0:N]; // Used to track which counter
                     // an output of annihilator_f connects to
reg     [N-1:0]      inhibitor_f; //Used to generate the inhibit signals
wire    [N-1:0]      inhibitor_f_wire;  // Wire for initial assignment
reg     [N-1:0]      inhibitor_fbar;
integer i;


//state parameters
localparam IDLE = 0;
localparam ACTIVE = 1;
localparam STALL = 2;
localparam FINISH = 3;

reg [1:0] state;

// Define parameters for the called modules - used for parameterization
defparam U1.n = n;
defparam U2.n = n;
defparam U3.n = n;

// Call One's count to determine number of 1's for input TT
Ones_Count U1 (TT_reg, inhibitor_f_wire);
// The transeunt triangles calculate the degrees for the computed
// annihilators - both f and f_bar
TranseuntTriangleToDegree U2 (TT_annihilator_f, f_degree_wire);
TranseuntTriangleToDegree U3 (TT_annihilator_fbar, f_bar_degree_wire);

always @(posedge CLK)
begin: statereg

    if (CLR)
        begin: Clearing
            TT_reg <= TT;
            state <= IDLE;
            AI <= 0;

            DONE <= 1'b0;

            Counter <= 1'b1; // Initially set to 1 to avoid the zero
                             // state, a known annihilator

            number_ones_fbar[0] <= 1'b0;
            number_ones_f[0] <= 1'b0;

            TT_annihilator_fbar <= 1'b0;
```

```
            TT_annihilator_f <= 1'b0;

            if(n % 2 == 1)  // Set Max to ceiling of n / 2
                begin
                    Max <= n/2 + 1;
                end
            else
                begin
                    Max <= n/2;
                end

        end // Clearing
    else
        begin
            case (state)
                // Indenting is shifted left inside each state
                // to allow the code to more properly fit on a printed
                // page

                // The state machine always begins in IDLE so this
                // state is used to perform some initialization that
                // threw off timing in the CLEAR state
                IDLE:
begin
    inhibitor_fbar <= 2**(N - inhibitor_f_wire); // Inhibits for fbar
    inhibitor_f <= 2**inhibitor_f_wire; //raise 2 ^ inhibitor.  This
                    // will be compared to counter to generate inhibit.

    // This for loop iteratively populates the number_ones_f and
    // number_ones_fbar registers, so that for each bit position
    // each register shows the number of that type that has been
    // encountered.  For example, if the input TT is, from least
    // significant bit to most significant bit, 1010, then
    // number_ones_f will have 01122, and number_ones_fbar will
    // have 00112. Each starts with 0 in the LSB. These registers
    // are used to apply the numbers generated by the counter to the
    // correct places in the input TT in order to annihilate the
    // function.  This is a blocking assignment because the code will
    // not work otherwise: each successive value depends on the
    // previous, so they can't all be assigned simultaneously.
    // There are fast ways to assign all the values simultaneously
    // (using multiple instances of one's cont) but it is not
    // necessary as the code simply runs too slow to work for
    // more than 4 variables.
    for (i = 0; i < N; i = i + 1)
        begin
            if(TT_reg[i] == 1'b0)
                begin:f
                    number_ones_f[i+1]      =  number_ones_f[i];
                    number_ones_fbar[i+1]   =  number_ones_fbar[i] + 1;
                end
            else
                begin:f_bar
                    number_ones_fbar[i+1]   =  number_ones_fbar[i];
                    number_ones_f[i+1]      =  number_ones_f[i] + 1;
```

```
                end
    end //for

    if(START)
        begin
            if(inhibitor_f_wire == N)
                begin
                    GlobalMinimum <= 0;  // Covers the all 1's case
                    state <= FINISH;
                end
            else
                begin
                    GlobalMinimum <= Max; // Initially set Global
                                // Minimum to the maximum possible AI
                    state <= ACTIVE;
                end
        end
end //IDLE
                // The ACTIVE state is responsible for the actual
                // determination of the AI.  Once it has enumerated
                // all possible states it exits to finish, leaving
                // the degree of the smallest annihilator in
                // GlobalMinimum
                ACTIVE:
begin
    Counter <= Counter + 1; // Must count each clock - this enumerates
                            // the annihilators

    // This series of if/else statements determines if either signal
    // should be inhibited (i.e. if we've checked all possible
    // annihilators for f/fbar based on the number of 1's/0's).
    // It also causes the state to change to the FINISH state
    // once both annihilator generators are inhibited.
    if(Counter >= inhibitor_f)
        begin
            if(Counter >= inhibitor_fbar) //All states are enumerated
                begin
                    state <= FINISH;
                end
            TT_annihilator_f_inhibit <= 1;
        end
    else
        begin
            TT_annihilator_f_inhibit <= 0;
        end

    if(Counter >= inhibitor_fbar)
        begin
            TT_annihilator_fbar_inhibit <= 1;
        end
    else
        begin
            TT_annihilator_fbar_inhibit <= 0;
        end
```

```
// This for loop is actually applying the appropriate inputs
// in order to annihilate the function being tested, based on the
// number_ones_f and fbar arrays that were populated in IDLE.
for (i = 0; i < N; i = i + 1)
    begin
        if(TT_reg[i] == 1'b0)
            begin:f_ann
                TT_annihilator_f[i] <=  1'b0;
                TT_annihilator_fbar[i] <=
                    Counter[number_ones_fbar[i]];
            end
        else
            begin:f_bar_ann
                TT_annihilator_fbar[i] <=  1'b0;
                TT_annihilator_f[i] <=  Counter[number_ones_f[i]];
            end
    end //for

// This long combination of if statements is simply to carry out
// a minimum function: it puts the minimum of 3 possible values
// into GlobalMinimum.  If GlobalMinimum is the smallest it remains
// unchanged.  Otherwise, the smaller of the degrees for the
// annihilators of f and fbar goes into GlobalMinimum, unless those
// signals are being inhibited (i.e. all possible annihilators
// have already been enumerated).  A more efficient method of
// comparison is possible, but this method is simple and completes
// within a clock period.
if(TT_annihilator_f_inhibit == 1'b0)
    begin
        if(f_degree_wire < GlobalMinimum)
            begin
                GlobalMinimum <= f_degree_wire;
            end
    end

if(TT_annihilator_fbar_inhibit == 1'b0)
    begin
        if(TT_annihilator_f_inhibit == 1'b0)
            begin
                if(f_bar_degree_wire < f_degree_wire)
                    begin
                        if(f_bar_degree_wire < GlobalMinimum)
                            begin
                                GlobalMinimum <= f_bar_degree_wire;
                            end
                    end
            end
        else
            begin
                if(f_bar_degree_wire < GlobalMinimum)
                    begin
                        GlobalMinimum <= f_bar_degree_wire;
                    end
                end
            end
```

```
end // ACTIVE
                // STALL isn't actually used, but could be used if
                // this was altered to function as a producer /
                // consumer model.  This state can be removed.
                STALL:
begin
    if(START)
        begin
            state <= ACTIVE;
        end

end //STALL
                // FINISH sets AI to the smallest degree annihilator
                // found and sets DONE causing control to return
                // to subr.mc
                FINISH:
begin
    AI <= GlobalMinimum;
    DONE <= 1'b1;
end //FINISH

            endcase

        end // state cases

end // statereg

//////////////////////////////////////////////////////////////////////

//Constant function to produce nbits_fact_n(n)
function integer ones_counter(input integer n);
integer m;
    begin
                ones_counter = 0;
                m = n;
                while (m > 0)
                    begin
                                ones_counter = ones_counter + m%2;
                                m = m >> 1;
                        end
        end
endfunction

//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//Constant function
function integer clogb2(input integer depth);
        begin
                for(clogb2=0; depth>0; clogb2 = clogb2 + 1)
                        depth = depth >> 1;
        end
endfunction

endmodule
```

A.3 SIMULTANEOUS EQUATION ALGORITHM SOURCE CODE (n = 4)

This algorithm enumerated all functions at n = 4 significantly faster than the brute force method.

1. main.c

```
//*********************************************************************
//
//  main.c  -    C program to run Algebraic_Immunity
//
//       Author:          Eric McCay
//       Created:         July 25, 2011
//
//       Description:  This program determines the Algebraic Immunity
//                     of all Boolean functions for a given n and
//                     provides an output specifying the number of
//                     functions with each AI.
//
//*********************************************************************

#include <map.h>
#include <stdlib.h>
#include <stdio.h>

void subr (int64_t*, int64_t*, int );

int main (int argc, char *argv[]) {
    FILE *res_map, *res_cpu;
        int mapnum = 0;
        int i;
        int64_t time_clock;
        int64_t *AI;

// Allocate array to hold the AI values
    AI = (int64_t *) malloc (4* sizeof (int64_t));

    for (i = 0; i < 4; i++){
            AI[i] = 0; // Zero out AI.
        }

    map_allocate (1); // Allocate the first map

    //This shows that the subr.mc has been called.  Subroutine
    //calls can take a considerable amount of time so this lets
    //the user know that execution has started properly.
    printf ("Calling subr.mc\n\n");

//  Call subroutine subr.mc on the MAP.
    subr (AI, &time_clock, mapnum);
```

```
    printf("Return from subr.mc\n\n");

// Print out the number of clocks.
    printf ("%lld clocks\n", time_clock);

// Print out the Algebraic Immunity of each Function
    printf("Listed below is the number of functions with each "
           "Algebraic Immunity\n\n");


        printf("AI = 3: %d\n",AI[3]);
        printf("AI = 2: %d\n",AI[2]);
        printf("AI = 1: %d\n",AI[1]);
        printf("AI = 0: %d\n",AI[0]);

    map_free (1); // Release the map

    exit(0);

    }//int main (int argc, char *argv[]) {
```

2. subr.mc

```
//*****************************************************************
//
//  subr.mc  - MAP C subroutine to determine Algebraic Immunity
//
//       Author:         Eric McCay
//       Created:        July 25, 2011
//
//       Description:  This program calls Algebraic_Immunity.v, which
//                     determines the Algebraic Immunity of the
//                     function provided in Truth Table Form.
//
//*****************************************************************

#include <libmap.h>
#define NUM 65536  //number of values in TT 2^(2^n)

void subr (int64_t ai[], int64_t *time, int mapnum) {

// Declare one OBM bank in the SRC-6 to store the number of
// functions with each possible AI value.
        OBM_BANK_B (AI, int64_t, 4)

        int64_t t0, t1; // Used to determine runtime
        int64_t my64bit_in; //input TT to test
        int64_t my64bit_out; //output AI of tested function
        int i, j, k, l, m, n;

        read_timer(&t0);

        for (i = 0; i < 4; i++)
```

```
        AI[i] = 0; //Initially, zero out the AI values
    k = 0;
    l = 0;
    m = 0;
    n = 0;

    //This for loop calls the macro file the required number
    //of times (65536 in this case) to determine the AI
    //for each possible TT input on 4 variables.  It then
    //uses a switch statement to tally the results for
    //each possible AI value.  For n=4, AI can be at most
    //two, so the case 3 statement never executes.
    //A modification for this subr.mc vice the one for the brute
    //force algorithm is that this one does not pass the all
    //0's or all 1's truth tables to the macro.  It is known that
    //AI is zero for these constant functions so they are not
    //tested, which simplified the macro design.
    for (i = 1; i < (NUM - 1); i++)
        {
            my64bit_in = i;
            my_operator (my64bit_in, &my64bit_out);
            j = my64bit_out;
            switch (j)
            {
            case 0:
                k++;
                break;
            case 1:
                l++;
                break;
            case 2:
                m++;
                break;
            case 3:
                n++;
                break;
            }

         }//for (i = 0; i < NUM; i++){

    AI[0] = 2; // It is known that there are 2 functions
               // with AI = 0 regardless of the number of variables
    AI[1] = l;
    AI[2] = m;
    AI[3] = n;

    read_timer(&t1);

    *time = (t1 - t0);

// Return AI values by DMAing TO the CPU
    DMA_CPU (OBM2CM, AI, MAP_OBM_stripe(1,"B"), ai,
            1, 4*sizeof(int64_t), 0);
    wait_DMA (0);
```

```
}
```

3. Algebraic_Immunity.v

```
module Algebraic_Immunity(TT, AI, DONE, CLK, CLR, START);
//----------------------------------------------------------------------
// Algebraic_Immunity -  Verilog code to determine the algebraic
//                       immunity of the provided function.
//
// Created:       August 24, 2011
// Author:        Eric McCay and Jon T. Butler
//
// Inputs:        TT - Truth table of the function being tested
// Outputs:       AlgebraicImmunity - The algebraic immunity of the
//                                    tested function
//
//----------------------------------------------------------------------
//

parameter       n = 4;       // The number of variables.
localparam      N = 2**n;    // Max number of elements in TT.

input           CLK;
input           CLR;
input           START;
input   [63:0]  TT;      // The function under test
wire    [63:0]  TT;

reg     [63:0]  TT_reg;

output  [63:0]  AI;      // Algebraic Immunity of function under test
output          DONE;    // Indicates completion
reg             DONE;
reg     [63:0]  AI;

integer i;

// Variables for simultaneous equation solving
// The SimultArray holds the simultaneous equations
// To solve.  It's structure for n=4 is:
//     A0 A1 A2 A3 A4 A1A2 A1A3 A1A4 A2A3 A2A4 A3A4
// g0  x  x  x  x  x  x    x    x    x    x    x
// g1  x  x  x  x  x  x    x    x    x    x    x
// g2  x  x  x  x  x  x    x    x    x    x    x
// g3  x  x  x  x  x  x    x    x    x    x    x
// g4  x  x  x  x  x  x    x    x    x    x    x
// g5  x  x  x  x  x  x    x    x    x    x    x
// g6  x  x  x  x  x  x    x    x    x    x    x
// g7  x  x  x  x  x  x    x    x    x    x    x
// g8  x  x  x  x  x  x    x    x    x    x    x
// g9  x  x  x  x  x  x    x    x    x    x    x
// g10 x  x  x  x  x  x    x    x    x    x    x
// g11 x  x  x  x  x  x    x    x    x    x    x
```

```
// g12 x  x  x  x  x  x    x    x    x    x    x
// g13 x  x  x  x  x  x    x    x    x    x    x
// g14 x  x  x  x  x  x    x    x    x    x    x
// g15 x  x  x  x  x  x    x    x    x    x    x

// The x's are filled in based on the TT that is input
// Where the x's are all 0's if the corresponding bit
// is 0 in the TT, and are filled in with appropriate
// values based on whether or not those terms appear
// for the that particular value in the TT.

reg    [4:0]        SimultArray [0:15]; // array to hold
                                        // the 2^n equations
reg    [15:0]       A0Array;
reg    [15:0]       A1Array;   // of each bit of the truth table
reg    [15:0]       A2Array;   // - it is used to create the
reg    [15:0]       A3Array;   // SimultArray.
reg    [15:0]       A4Array;

reg    [n:0]        RowCounter; // Keeps track of row being searched
reg    [n:0]        RowUpdate;  //Used to maintain the desired position
                                //of the next row of interest
reg    [n:0]        ColCounter; // Tracks the column of interest

reg    [4:0]        Row0Terms;  // Used for determining if the
reg    [4:0]        Row1Terms;  // annihilator is of degree 1
reg    [4:0]        Row2Terms;
reg    [4:0]        Row3Terms;

reg                 CompTrack;  // Used to track if complement has
                                // been checked

//state parameters
localparam Idle = 0;
localparam Init = 1;
localparam RowSearch = 2;
localparam RowFound = 3;
localparam RowSwap = 4;
localparam FindDegree = 5;
localparam UpdateDegree = 6;
localparam ComplementCheck = 7;
localparam Finish = 8;

reg [3:0] state;

always @(posedge CLK)
begin: statereg

    if (CLR)
        begin: Clearing

            state <= Idle;
            TT_reg <= TT;

            AI <= 0;
```

```
            DONE <= 1'b0;

            CompTrack <= 0;

            // Initialize the SimultArray to all 0's
            // And put the correct values in the
            // Arrays used to build SimultArray
            for(i = 0;i < N; i = i+1)
                begin
                    A0Array[i] <= 1;
                    A1Array[i] <= i%2;
                    A2Array[i] <= (i>>1)%2;
                    A3Array[i] <= (i>>2)%2;
                    A4Array[i] <= (i>>3)%2;
                    SimultArray[i] <= {5{1'b0}};
                end

        end // Clearing
    else
        begin
            case (state)
                // The states are indented left to increase readability

                // Idle waits for start and then moves flow to the Init
                // state - it performs no operations so that the same
                // pathway can be used for the function and its
                // complement.
                Idle:
begin
    if(START)
        begin
            state <= Init;
        end
end //Idle

                //Init builds the SimultArray, which is then solved
                //to determine the lowest degree annihilator for the
                //function being tested.  It also initializes all
                //variables used to process the array.
                Init:
begin

    for(i = 0;i < N; i = i+1)
        begin
            SimultArray[i] <= {TT_reg[i]&A4Array[i],
                               TT_reg[i]&A3Array[i],
                               TT_reg[i]&A2Array[i],
                               TT_reg[i]&A1Array[i],
                               TT_reg[i]&A0Array[i]};

        end

    RowCounter <= 0;
    RowUpdate <= 0;
```

```
    ColCounter <= 0;

    state <= RowSearch;

end // Init

                // This state checks the rows 1 at a time, starting
                // after the last row to have been updated, and
                // attempts to find a 1 in the column of interest.
                RowSearch:
begin
    // If this code executes, we have established
    // reduced row echelon form and are ready
    // to determine the lowest degree annihilator
    if(ColCounter == 5)
        begin
            Row0Terms <= SimultArray[0];
            Row1Terms <= SimultArray[1];
            Row2Terms <= SimultArray[2];
            Row3Terms <= SimultArray[3];
            state <= FindDegree;
        end
    // If the next code executes (meaning we have counted
    // all rows) then AI is 1 because we have at least
    // one free variable, allowing us to produce
    // a degree 1 annihilator
    else if(RowCounter == N)
        begin
            AI <= 1;
            state <= Finish;
        end
    // This executes if we find a 1 in the column of interest
    else if(((SimultArray[RowCounter]>>(4-ColCounter))%2) == 1)
        begin
            state <= RowFound;
        end
    // The default code moves us to the next row to continue looking
    else
        begin
            RowCounter <= RowCounter + 1;
        end

end // RowSearch

                // Entering this state means we found a row with a 1
                // in the column of interest.  We will use this row to
                //zero out the column of interest in all other rows.
                RowFound:
begin

    // This for loop adds (xors) the found row with all other
    // rows that have a 1 in the column of interest to zero
    // them out.
    for(i = 0;i < N; i = i + 1)
        begin
```

```
            if((((SimultArray[i]>>(4-ColCounter))%2) == 1)
              &&(i != RowCounter))
                begin
                    SimultArray[i] <= (SimultArray[i] ^
                                       SimultArray[RowCounter]);
                end
        end

    if (RowCounter == RowUpdate)//If true the row is in the right place
        begin
            RowUpdate <= RowUpdate + 1;
            ColCounter <= ColCounter + 1;
            state <= RowSearch;
            RowCounter <= RowUpdate + 1;
        end
    else // the row is in the wrong place
        begin
            ColCounter <= ColCounter + 1;
            state <= RowSwap;
        end

end // RowFound

                // Swaps the found row to the correct position
                RowSwap:
begin

    SimultArray[RowCounter] <= SimultArray[RowUpdate];
    SimultArray[RowUpdate] <= SimultArray[RowCounter];

    RowUpdate <= RowUpdate + 1;
    RowCounter <= RowUpdate + 1;

    state <= RowSearch;

end // RowSwap

                // Counts the number of 1's in each row
                FindDegree:
begin

    Row0Terms <= (Row0Terms[4]+Row0Terms[3]+Row0Terms[2]+Row0Terms[1]
                  +Row0Terms[0]);
    Row1Terms <= (Row1Terms[4]+Row1Terms[3]+Row1Terms[2]+Row1Terms[1]
                  +Row1Terms[0]);
    Row2Terms <= (Row2Terms[4]+Row2Terms[3]+Row2Terms[2]+Row2Terms[1]
                  +Row2Terms[0]);
    Row3Terms <= (Row3Terms[4]+Row3Terms[3]+Row3Terms[2]+Row3Terms[1]
                  +Row3Terms[0]);

    state <= UpdateDegree;

end // FindDegree

                // First, it checks to see if any row had two 1's. If
```

```
                // so, AI is 1 (those two variables combine to form
                // a degree one annihilator).  Otherwise, it checks
                // to see if the complement has been tested.  If not,
                // we go test the complement.  If the complement has
                // been tested, this function has no degree one
                // annihilators and so AI=2.
                UpdateDegree:
begin
    if((Row0Terms>1)||(Row1Terms>1)||(Row2Terms>1)||(Row3Terms>1))
        begin
            AI <= 1;
            state <= Finish;
        end
    else
        begin
            if(CompTrack == 0)
                begin
                    state <= ComplementCheck;
                end
            else
                begin
                    AI <= 2;
                    state <= Finish;
                end
        end

end //UpdateDegree

                // This complements the truth table, updates to show
                // that we are now testing the complement, and starts
                // the process over back at Init.
                ComplementCheck:
begin
    TT_reg <= ~TT;
    state <= Init;
    CompTrack <= 1;
end //ComplementCheck

                // Testing is complete.  The AI has already been set,
                // so the macro just exits
                Finish:
begin
    DONE <= 1'b1;
end //Finish

            endcase

        end // state cases

end // statereg

endmodule
```

A.4 SIMULTANEOUS EQUATION ALGORITHM SOURCE CODE (n = 5)

This algorithm was the first known to compute AI for all functions for n = 5. It is an extension of the algorithm used for n = 4.

1. main.c

```
//*********************************************************************
//
//  main.c  -    C program to run Algebraic_Immunity
//
//       Author:          Eric McCay
//       Created:         July 25, 2011
//
//       Description:  This program determines the Algebraic Immunity
//                     of all Boolean functions for a given n and
//                     provides an output specifying the number of
//                     functions with each AI.
//
//*********************************************************************

#include <map.h>
#include <stdlib.h>
#include <stdio.h>
#define NUM 4294967296  //number of values in TT 2^(2^n)

void subr (uint64_t*, uint64_t*, int );

// Use uint64_t for all variables in this function because of the
// large number we are counting to (2^32).
int main (int argc, char *argv[]) {
    FILE *res_map, *res_cpu;
        int mapnum = 0; // use map 0
        uint64_t i;
        uint64_t time_clock; // used for timing
        uint64_t *AI;

// Allocate array of AI values
    AI = (uint64_t *) malloc (4* sizeof (uint64_t));

        for (i = 0; i < 4; i++){
                AI[i] = 0; //Zero out AI.
            }

    map_allocate (1); // hold the map

    //This shows that the subr.mc has been called.  Subroutine
    //calls can take a considerable amount of time so this lets
    //the user know that execution has started properly.
    printf ("Calling subr.mc\n\n");

//  Call subroutine subr.mc on the MAP.
    subr (AI, &time_clock, mapnum);
```

```
    printf("Return from subr.mc\n\n");

// Print out the number of clocks.
printf ("%lld clocks\n", time_clock);

// Print out the Algebraic Immunity of each Function
    printf("Listed below is the number of functions with each "
           "Algebraic Immunity\n\n");

        printf("AI = 3: %lld\n",AI[3]);
        printf("AI = 2: %lld\n",AI[2]);
        printf("AI = 1: %lld\n",AI[1]);
        printf("AI = 0: %lld\n",AI[0]);

    map_free (1); // release the map

    exit(0);

    }//int main (int argc, char *argv[]) {
```

2. subr.mc

```
//*****************************************************************
//
//  subr.mc  - MAP C subroutine to determine Algebraic Immunity
//
//       Author:         Eric McCay
//       Created:        July 25, 2011
//
//       Description:  This program calls Algebraic_Immunity.v, which
//                     determines the Algebraic Immunity of the
//                     function provided in Truth Table Form.
//
//*****************************************************************

#include <libmap.h>
#define NUM 4294967296  //number of values in TT 2^(2^n)

// all variables in this function are declared as uint64_t due to
// large numbers being worked with (up to 2^32)
void subr (uint64_t ai[], uint64_t *time, int mapnum) {

// Declare one OBM bank in the SRC-6 to store the number of
// functions with each possible AI value.
        OBM_BANK_B (AI, uint64_t, 4)

        uint64_t t0, t1; // used to determine runtime
        uint64_t my64bit_in; // TT  of function being tested
        uint64_t my64bit_out; // AI of tested function
        uint64_t i, j, k, l, m, n;

        read_timer(&t0);
```

```
        for (i = 0; i < 4; i++)
            AI[i] = 0; // set AI to 0 initially
        k = 0;
        l = 0;
        m = 0;
        n = 0;

        //This for loop calls the macro file the required number
        //of times (4294967296 in this case) to determine the AI
        //for each possible TT input on 5 variables.  It then
        //uses a switch statement to tally the results for
        //each possible AI value.
        //A modification for this subr.mc vice the one for the brute
        //force algorithm is that this one does not pass the all
        //0's or all 1's truth tables to the macro.  It is known that
        //AI is zero for these constant functions so they are not
        //tested, which simplified the macro design.
        for (i = 1; i < (NUM - 1); i++)
            {
                 my64bit_in = i;
                 my_operator (my64bit_in, &my64bit_out);
                 j = my64bit_out;
                 switch (j)
                 {
                 case 0:
                     k++;
                     break;
                 case 1:
                     l++;
                     break;
                 case 2:
                     m++;
                     break;
                 case 3:
                     n++;
                     break;
                 }

             }

        AI[0] = 2;
        AI[1] = l;
        AI[2] = m;
        AI[3] = n;

        read_timer(&t1);

        *time = (t1 - t0);

// Return AI values by DMAing TO the CPU
        DMA_CPU (OBM2CM, AI, MAP_OBM_stripe(1,"B"), ai,
            1, 4*sizeof(uint64_t), 0);
        wait_DMA (0);
```

```
}
```

3. Algebraic_Immunity.v

```
module Algebraic_Immunity(TT, AI, DONE, CLK, CLR, START);
//----------------------------------------------------------------------
// Algebraic_Immunity -  Verilog code to determine the algebraic
//                       immunity of the provided function.
//
// Created:       August 24, 2011
// Author:        Eric McCay and Jon T. Butler
//
// Inputs:        TT - Truth table of the function being tested
// Outputs:       AlgebraicImmunity - The algebraic immunity of the
//                                    tested function
//
//----------------------------------------------------------------------
//

parameter       n = 5;     // The number of variables.
localparam      N = 2**n;  // Max number of elements in TT.

input           CLK;
input           CLR;
input           START;
input   [63:0]  TT;        // The function under test
wire    [63:0]  TT;

reg     [63:0]  TT_reg;

output  [63:0]  AI;        // Algebraic Immunity of function under test
output          DONE;      // Indicates completion
reg             DONE;
reg     [63:0]  AI;

integer i;

reg    [15:0]       SimultArray [0:31]; // array to hold
                                        // the 2^n equations
reg    [31:0]       A0Array;
reg    [31:0]       A1Array;   // of each bit of the truth table
reg    [31:0]       A2Array;   // - it is used to create the
reg    [31:0]       A3Array;   // SimultArray.
reg    [31:0]       A4Array;
reg    [31:0]       A5Array;

reg    [n:0]        RowCounter; // Keeps track of row being searched
reg    [n:0]        RowUpdate;  //Used to maintain the desired position
                                // of the next row of interest
reg    [n:0]        ColCounter; // Tracks the column of interest

reg    [15:0]       Row0Terms1; // Used for determining if the
reg    [15:0]       Row1Terms1; // annihilator is of degree 1
```

```
reg     [15:0]        Row2Terms1;
reg     [15:0]        Row3Terms1;
reg     [15:0]        Row4Terms1;
reg     [15:0]        Row0Terms2;
reg     [15:0]        Row1Terms2;
reg     [15:0]        Row2Terms2;
reg     [15:0]        Row3Terms2;
reg     [15:0]        Row4Terms2;
reg     [15:0]        Row5Terms2;
reg     [15:0]        Row6Terms2;
reg     [15:0]        Row7Terms2;
reg     [15:0]        Row8Terms2;
reg     [15:0]        Row9Terms2;
reg     [15:0]        Row10Terms2;
reg     [15:0]        Row11Terms2;
reg     [15:0]        Row12Terms2;
reg     [15:0]        Row13Terms2;
reg     [15:0]        Row14Terms2;
reg     [15:0]        Row15Terms2;

reg                   CompTrack;  // Used to determine if complement
                                  // has been checked

//state parameters
localparam Idle = 0;
localparam Init = 1;
localparam RowSearch1 = 2;
localparam RowFound1 = 3;
localparam RowSwap1 = 4;
localparam FindDegree1 = 5;
localparam UpdateDegree1 = 6;
localparam ComplementCheck = 7;
localparam RowSearch2 = 8;
localparam RowFound2 = 9;
localparam RowSwap2 = 10;
localparam FindDegree2 = 11;
localparam UpdateDegree2 = 12;
localparam Finish = 13;

reg [3:0] state;

always @(posedge CLK)
begin: statereg

    if (CLR)
        begin: Clearing

            state <= Idle;
            TT_reg <= TT;

            AI <= 0;

            DONE <= 1'b0;

            CompTrack <= 0;
```

```
            // Initialize the SimultArray to all 0's
            // And put the correct values in the
            // Arrays used to build SimultArray
            for(i = 0;i < N; i = i+1)
                begin
                    A0Array[i] <= 1;
                    A1Array[i] <= i%2;
                    A2Array[i] <= (i>>1)%2;
                    A3Array[i] <= (i>>2)%2;
                    A4Array[i] <= (i>>3)%2;
                    A5Array[i] <= (i>>4)%2;
                    SimultArray[i] <= {16{1'b0}};
                end

        end // Clearing
    else
        begin
            case (state)
                // The state code is shifted left for readability

                // Idle waits for start and then moves flow to the Init
                // state - it performs no operations so that the same
                // pathway can be used for the function and its
                // complement.
                Idle:
begin
    if(START)
        begin
            state <= Init;
        end
end //Idle

                //Init builds the SimultArray, which is then solved
                //to determine the lowest degree annihilator for the
                //function being tested.  It also initializes all
                //variables used to process the array.
                Init:
begin

    for(i = 0;i < N; i = i+1)
        begin
            SimultArray[i] <= {TT_reg[i]&A4Array[i]&A5Array[i],
                               TT_reg[i]&A3Array[i]&A5Array[i],
                               TT_reg[i]&A3Array[i]&A4Array[i],
                               TT_reg[i]&A2Array[i]&A5Array[i],
                               TT_reg[i]&A2Array[i]&A4Array[i],
                               TT_reg[i]&A2Array[i]&A3Array[i],
                               TT_reg[i]&A1Array[i]&A5Array[i],
                               TT_reg[i]&A1Array[i]&A4Array[i],
                               TT_reg[i]&A1Array[i]&A3Array[i],
                               TT_reg[i]&A1Array[i]&A2Array[i],
                               TT_reg[i]&A5Array[i],
                               TT_reg[i]&A4Array[i],
                               TT_reg[i]&A3Array[i],
```

```
                                TT_reg[i]&A2Array[i],
                                TT_reg[i]&A1Array[i],
                                TT_reg[i]&A0Array[i]};

        end

                            RowCounter <= 0;
                            RowUpdate <= 0;
                            ColCounter <= 0;

                            state <= RowSearch1;

                        end // Init

                    // This state checks the rows 1 at a time, starting
                    // after the last row to have been updated, and
                    // attempts to find a 1 in the column of interest.
                    RowSearch1:
begin
    // If this code executes, we have established
    // reduced row echelon form and are ready
    // to determine the lowest degree annihilator
    if(ColCounter == 6)
        begin
            Row0Terms1 <= SimultArray[0];
            Row1Terms1 <= SimultArray[1];
            Row2Terms1 <= SimultArray[2];
            Row3Terms1 <= SimultArray[3];
            Row4Terms1 <= SimultArray[4];
            state <= FindDegree1;
        end
    // If the next code executes (meaning we have counted
    // all rows) then AI is 1 because we have at least
    // one free variable, allowing us to produce
    // a degree 1 annihilator
    else if(RowCounter == N)
        begin
            AI <= 1;
            state <= Finish;
        end
    // This executes if we find a 1 in the column of interest
    else if(((SimultArray[RowCounter]>>(ColCounter))%2) == 1)
        begin
            state <= RowFound1;
        end
    // The default code moves us to the next row to continue looking
    else
        begin
            RowCounter <= RowCounter + 1;
        end

end // RowSearch1

                    // Entering this state means we found a row with a 1 in
                    // the column of interest.  We will use this row to
```

```
                // zero out the column of interest in all other rows.
                RowFound1:
begin

    // This for loop adds (xors) the found row with all other
    // rows that have a 1 in the column of interest to zero
    // them out.
    for(i = 0;i < N; i = i + 1)
        begin
            if((((SimultArray[i]>>(ColCounter))%2) == 1)
                  &&(i != RowCounter))
                begin
                    SimultArray[i] <= (SimultArray[i] ^
                                       SimultArray[RowCounter]);
                end
        end

    if (RowCounter == RowUpdate)//If true the row is in the right place
        begin
            RowUpdate <= RowUpdate + 1;
            ColCounter <= ColCounter + 1;
            state <= RowSearch1;
            RowCounter <= RowUpdate + 1;
        end
    else // the row is in the wrong place
        begin
            ColCounter <= ColCounter + 1;
            state <= RowSwap1;
        end

                    end // RowFound1

                // Swaps the found row to the correct position
                RowSwap1:
begin

    SimultArray[RowCounter] <= SimultArray[RowUpdate];
    SimultArray[RowUpdate] <= SimultArray[RowCounter];

    RowUpdate <= RowUpdate + 1;
    RowCounter <= RowUpdate + 1;

    state <= RowSearch1;

end // RowSwap1

                // Counts the number of 1's in each row
                FindDegree1:
begin

    Row0Terms1 <= (Row0Terms1[5]+Row0Terms1[4]+Row0Terms1[3]
                   +Row0Terms1[2]+Row0Terms1[1]+Row0Terms1[0]);
    Row1Terms1 <= (Row1Terms1[5]+Row1Terms1[4]+Row1Terms1[3]
                   +Row1Terms1[2]+Row1Terms1[1]+Row1Terms1[0]);
    Row2Terms1 <= (Row2Terms1[5]+Row2Terms1[4]+Row2Terms1[3]
```

```
                    +Row2Terms1[2]+Row2Terms1[1]+Row2Terms1[0]);
    Row3Terms1 <= (Row3Terms1[5]+Row3Terms1[4]+Row3Terms1[3]
                    +Row3Terms1[2]+Row3Terms1[1]+Row3Terms1[0]);
    Row4Terms1 <= (Row4Terms1[5]+Row4Terms1[4]+Row4Terms1[3]
                    +Row4Terms1[2]+Row4Terms1[1]+Row4Terms1[0]);

    state <= UpdateDegree1;

end // FindDegree1

                // First, it checks to see if any row had two 1's. If
                // so, AI is 1 (those two variables combine to form
                // a degree one annihilator).  Otherwise, it goes to
                // RowSearch2, which is looking for a degree 2
                // annihilator
                UpdateDegree1:
begin
    if((Row0Terms1>1)||(Row1Terms1>1)||(Row2Terms1>1)||(Row3Terms1>1)
      ||(Row4Terms1>1))
        begin
            AI <= 1;
            state <= Finish;
        end
    else
        begin
            state <= RowSearch2;
        end

end //UpdateDegree1

                // This is similar to RowSearch1, except this is
                // looking for degree 2 annihilators
                RowSearch2:
begin
    // If this code executes, we have established
    // reduced row echelon form and are ready
    // to determine the lowest degree annihilator
    if(ColCounter == 16)
        begin
            Row0Terms2 <= SimultArray[0];
            Row1Terms2 <= SimultArray[1];
            Row2Terms2 <= SimultArray[2];
            Row3Terms2 <= SimultArray[3];
            Row4Terms2 <= SimultArray[4];
            Row5Terms2 <= SimultArray[5];
            Row6Terms2 <= SimultArray[6];
            Row7Terms2 <= SimultArray[7];
            Row8Terms2 <= SimultArray[8];
            Row9Terms2 <= SimultArray[9];
            Row10Terms2 <= SimultArray[10];
            Row11Terms2 <= SimultArray[11];
            Row12Terms2 <= SimultArray[12];
            Row13Terms2 <= SimultArray[13];
            Row14Terms2 <= SimultArray[14];
            Row15Terms2 <= SimultArray[15];
```

```
            state <= FindDegree2;
        end
    // If the next code executes (meaning we have counted
    // all rows) then AI is 2 because we have at least
    // one free variable, allowing us to produce
    // a degree 2 annihilator
    else if(RowCounter == N)
        begin
            AI <= 2;
            state <= ComplementCheck;
        end
    else if(((SimultArray[RowCounter]>>(ColCounter))%2) == 1)
        begin
            state <= RowFound2;
        end
    else
        begin
            RowCounter <= RowCounter + 1;
        end

end // RowSearch2

                // Entering this state means we found a row with a 1
                // in the column of interest.  We will use this row to
                // zero out the column of interest in all other rows.
                RowFound2:
begin

    // This for loop adds (xors) the found row with all other
    // rows that have a 1 in the column of interest to zero
    // them out.
    for(i = 0;i < N; i = i + 1)
        begin
            if((((SimultArray[i]>>(ColCounter))%2) == 1)
              &&(i != RowCounter))
                begin
                    SimultArray[i] <= (SimultArray[i] ^
                                       SimultArray[RowCounter]);
                end
            end

    if (RowCounter == RowUpdate)//If true the row is in the right place
        begin
            RowUpdate <= RowUpdate + 1;
            ColCounter <= ColCounter + 1;
            state <= RowSearch2;
            RowCounter <= RowUpdate + 1;
        end
    else // row in the wrong place
        begin
            ColCounter <= ColCounter + 1;
            state <= RowSwap2;
        end

end // RowFound2
```

```
                // swap the row to the correct place
                RowSwap2:
begin

    SimultArray[RowCounter] <= SimultArray[RowUpdate];
    SimultArray[RowUpdate] <= SimultArray[RowCounter];

    RowUpdate <= RowUpdate + 1;
    RowCounter <= RowUpdate + 1;

    state <= RowSearch2;

end // RowSwap2

                // Counts the number of 1's in each row
                FindDegree2:
begin

    Row0Terms2 <= (Row0Terms2[15]+Row0Terms2[14]+Row0Terms2[13]
        +Row0Terms2[12]+Row0Terms2[11]+Row0Terms2[10]+Row0Terms2[9]
        +Row0Terms2[8]+Row0Terms2[7]+Row0Terms2[6]+Row0Terms2[5]
        +Row0Terms2[4]+Row0Terms2[3]+Row0Terms2[2]+Row0Terms2[1]
        +Row0Terms2[0]);
    Row1Terms2 <= (Row1Terms2[15]+Row1Terms2[14]+Row1Terms2[13]
        +Row1Terms2[12]+Row1Terms2[11]+Row1Terms2[10]+Row1Terms2[9]
        +Row1Terms2[8]+Row1Terms2[7]+Row1Terms2[6]+Row1Terms2[5]
        +Row1Terms2[4]+Row1Terms2[3]+Row1Terms2[2]+Row1Terms2[1]
        +Row1Terms2[0]);
    Row2Terms2 <= (Row2Terms2[15]+Row2Terms2[14]+Row2Terms2[13]
        +Row2Terms2[12]+Row2Terms2[11]+Row2Terms2[10]+Row2Terms2[9]
        +Row2Terms2[8]+Row2Terms2[7]+Row2Terms2[6]+Row2Terms2[5]
        +Row2Terms2[4]+Row2Terms2[3]+Row2Terms2[2]+Row2Terms2[1]
        +Row2Terms2[0]);
    Row3Terms2 <= (Row3Terms2[15]+Row3Terms2[14]+Row3Terms2[13]
        +Row3Terms2[12]+Row3Terms2[11]+Row3Terms2[10]+Row3Terms2[9]
        +Row3Terms2[8]+Row3Terms2[7]+Row3Terms2[6]+Row3Terms2[5]
        +Row3Terms2[4]+Row3Terms2[3]+Row3Terms2[2]+Row3Terms2[1]
        +Row3Terms2[0]);
    Row4Terms2 <= (Row4Terms2[15]+Row4Terms2[14]+Row4Terms2[13]
        +Row4Terms2[12]+Row4Terms2[11]+Row4Terms2[10]+Row4Terms2[9]
        +Row4Terms2[8]+Row4Terms2[7]+Row4Terms2[6]+Row4Terms2[5]
        +Row4Terms2[4]+Row4Terms2[3]+Row4Terms2[2]+Row4Terms2[1]
        +Row4Terms2[0]);
    Row5Terms2 <= (Row5Terms2[15]+Row5Terms2[14]+Row5Terms2[13]
        +Row5Terms2[12]+Row5Terms2[11]+Row5Terms2[10]+Row5Terms2[9]
        +Row5Terms2[8]+Row5Terms2[7]+Row5Terms2[6]+Row5Terms2[5]
        +Row5Terms2[4]+Row5Terms2[3]+Row5Terms2[2]+Row5Terms2[1]
        +Row5Terms2[0]);
    Row6Terms2 <= (Row6Terms2[15]+Row6Terms2[14]+Row6Terms2[13]
        +Row6Terms2[12]+Row6Terms2[11]+Row6Terms2[10]+Row6Terms2[9]
        +Row6Terms2[8]+Row6Terms2[7]+Row6Terms2[6]+Row6Terms2[5]
        +Row6Terms2[4]+Row6Terms2[3]+Row6Terms2[2]+Row6Terms2[1]
        +Row6Terms2[0]);
    Row7Terms2 <= (Row7Terms2[15]+Row7Terms2[14]+Row7Terms2[13]
```

```
        +Row7Terms2[12]+Row7Terms2[11]+Row7Terms2[10]+Row7Terms2[9]
        +Row7Terms2[8]+Row7Terms2[7]+Row7Terms2[6]+Row7Terms2[5]
        +Row7Terms2[4]+Row7Terms2[3]+Row7Terms2[2]+Row7Terms2[1]
        +Row7Terms2[0]);
    Row8Terms2 <= (Row8Terms2[15]+Row8Terms2[14]+Row8Terms2[13]
        +Row8Terms2[12]+Row8Terms2[11]+Row8Terms2[10]+Row8Terms2[9]
        +Row8Terms2[8]+Row8Terms2[7]+Row8Terms2[6]+Row8Terms2[5]
        +Row8Terms2[4]+Row8Terms2[3]+Row8Terms2[2]+Row8Terms2[1]
        +Row8Terms2[0]);
    Row9Terms2 <= (Row9Terms2[15]+Row9Terms2[14]+Row9Terms2[13]
        +Row9Terms2[12]+Row9Terms2[11]+Row9Terms2[10]+Row9Terms2[9]
        +Row9Terms2[8]+Row9Terms2[7]+Row9Terms2[6]+Row9Terms2[5]
        +Row9Terms2[4]+Row9Terms2[3]+Row9Terms2[2]+Row9Terms2[1]
        +Row9Terms2[0]);
    Row10Terms2 <= (Row10Terms2[15]+Row10Terms2[14]+Row10Terms2[13]
        +Row10Terms2[12]+Row10Terms2[11]+Row10Terms2[10]+Row10Terms2[9]
        +Row10Terms2[8]+Row10Terms2[7]+Row10Terms2[6]+Row10Terms2[5]
        +Row10Terms2[4]+Row10Terms2[3]+Row10Terms2[2]+Row10Terms2[1]
        +Row10Terms2[0]);
    Row11Terms2 <= (Row11Terms2[15]+Row11Terms2[14]+Row11Terms2[13]
        +Row11Terms2[12]+Row11Terms2[11]+Row11Terms2[10]+Row11Terms2[9]
        +Row11Terms2[8]+Row11Terms2[7]+Row11Terms2[6]+Row11Terms2[5]
        +Row11Terms2[4]+Row11Terms2[3]+Row11Terms2[2]+Row11Terms2[1]
        +Row11Terms2[0]);
    Row12Terms2 <= (Row12Terms2[15]+Row12Terms2[14]+Row12Terms2[13]
        +Row12Terms2[12]+Row12Terms2[11]+Row12Terms2[10]+Row12Terms2[9]
        +Row12Terms2[8]+Row12Terms2[7]+Row12Terms2[6]+Row12Terms2[5]
        +Row12Terms2[4]+Row12Terms2[3]+Row12Terms2[2]+Row12Terms2[1]
        +Row12Terms2[0]);
    Row13Terms2 <= (Row13Terms2[15]+Row13Terms2[14]+Row13Terms2[13]
        +Row13Terms2[12]+Row13Terms2[11]+Row13Terms2[10]+Row13Terms2[9]
        +Row13Terms2[8]+Row13Terms2[7]+Row13Terms2[6]+Row13Terms2[5]
        +Row13Terms2[4]+Row13Terms2[3]+Row13Terms2[2]+Row13Terms2[1]
        +Row13Terms2[0]);
    Row14Terms2 <= (Row14Terms2[15]+Row14Terms2[14]+Row14Terms2[13]
        +Row14Terms2[12]+Row14Terms2[11]+Row14Terms2[10]+Row14Terms2[9]
        +Row14Terms2[8]+Row14Terms2[7]+Row14Terms2[6]+Row14Terms2[5]
        +Row14Terms2[4]+Row14Terms2[3]+Row14Terms2[2]+Row14Terms2[1]
        +Row14Terms2[0]);
    Row15Terms2 <= (Row15Terms2[15]+Row15Terms2[14]+Row15Terms2[13]
        +Row15Terms2[12]+Row15Terms2[11]+Row15Terms2[10]+Row15Terms2[9]
        +Row15Terms2[8]+Row15Terms2[7]+Row15Terms2[6]+Row15Terms2[5]
        +Row15Terms2[4]+Row15Terms2[3]+Row15Terms2[2]+Row15Terms2[1]
        +Row15Terms2[0]);


    state <= UpdateDegree2;

end // FindDegree2

                // First, it checks if there are two 1's in a row,
                // indicating a degree 2 annihilator.  If so, it goes
                // to ComplementCheck.  If not, it checks if the
                // complement has been tested.  If so and AI is still
                // set to 0, then AI is 3 (i.e. neither the function
```

```
                // nor its complement had a degree 2 or lower
                // annihilator.  Otherwise, it checks to see if this
                // is the complement and AI is currently set to 2.
                // If it is, we are done testing and AI is 2.  Default
                // is to go to ComplementCheck without adjusting AI,
                // which signals the the original function is being
                // tested and has no annihilator less then degree 3.
                UpdateDegree2:
begin
    if((Row0Terms2>1)||(Row1Terms2>1)||(Row2Terms2>1)||(Row3Terms2>1)
      ||(Row4Terms2>1)||(Row5Terms2>1)||(Row6Terms2>1)||(Row7Terms2>1)
      ||(Row8Terms2>1)||(Row9Terms2>1)||(Row10Terms2>1)
      ||(Row11Terms2>1)||(Row12Terms2>1)||(Row13Terms2>1)
      ||(Row14Terms2>1)||(Row15Terms2>1))
        begin
            AI <= 2;
            state <= ComplementCheck;
        end
    else
        begin
            if((CompTrack == 1)&&(AI == 0))
                begin
                    AI <= 3;
                    state <= Finish;
                end
            else if((CompTrack == 1)&&(AI == 2))
                begin
                    state <= Finish;
                end
            else
                begin
                    state <= ComplementCheck;
                end
        end

end //UpdateDegree2

                // If AI is set to 2 and the complement has been
                // tested, we are done checking and can exit.
                // Otherwise, it complements the TT, sets the tracker
                // and starts back over at Init to test the complement.
                ComplementCheck:
begin
    if((AI == 2)&&(CompTrack == 1))
        begin
            state <= Finish;
        end
    else if(CompTrack == 0)
        begin
            TT_reg <= ~TT;
            state <= Init;
            CompTrack <= 1;
        end
end //ComplementCheck
```

```
                // Testing complete.  Set DONE and exit.
                Finish:
begin
    DONE <= 1'b1;
end //Finish

            endcase

        end // state cases

end // statereg

endmodule
```

A.5 SIMULTANEOUS EQUATION ALGORITHM SOURCE CODE (n = 6)

This algorithm was used to perform Monte Carlo trials to estimate the distribution of functions with various algebraic immunities for n = 6. It is an extension of the algorithm used for n = 5.

1. main.c

```
//*********************************************************************
//
//  main.c  -    C program to run Algebraic_Immunity
//
//       Author:         Eric McCay
//       Created:        July 25, 2011
//
//       Description:  This program determines the Algebraic Immunity
//                     of all Boolean functions for a given n and
//                     provides an output specifying the number of
//                     functions with each AI.
//
//*********************************************************************

/*
A C-program for MT19937-64 (2004/9/29 version).
Coded by Takuji Nishimura and Makoto Matsumoto.

This is a 64-bit version of Mersenne Twister pseudorandom number
generator.

Before using, initialize the state by using init_genrand64(seed)
or init_by_array64(init_key, key_length).

Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
```

```
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote
   products derived from this software without specific prior written
   permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

   References:
   T. Nishimura, ``Tables of 64-bit Mersenne Twisters''
     ACM Transactions on Modeling and
     Computer Simulation 10. (2000) 348--357.
   M. Matsumoto and T. Nishimura,
     ``Mersenne Twister: a 623-dimensionally equidistributed
       uniform pseudorandom number generator''
     ACM Transactions on Modeling and
     Computer Simulation 8. (Jan. 1998) 3--30.

   Any feedback is very welcome.
   http://www.math.hiroshima-u.ac.jp/~m-mat/MT/emt.html
   email: m-mat @ math.sci.hiroshima-u.ac.jp (remove spaces)
*/

#include <map.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define NUM 500000000  //number of iterations to perform

#define NN 312
#define MM 156
#define MATRIX_A 0xB5026F5AA96619E9ULL
#define UM 0xFFFFFFFF80000000ULL /* Most significant 33 bits */
#define LM 0x7FFFFFFFULL /* Least significant 31 bits */

void subr (uint64_t*, uint64_t*, int );
```

```c
/* The array for the state vector */
static unsigned long long mt[NN];
/* mti==NN+1 means mt[NN] is not initialized */
static int mti=NN+1;

/* initializes mt[NN] with a seed */
void init_genrand64(unsigned long long seed)
{
    mt[0] = seed;
    for (mti=1; mti<NN; mti++)
        mt[mti] =  (6364136223846793005ULL * (mt[mti-1] ^
                    (mt[mti-1] >> 62)) + mti);
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
void init_by_array64(unsigned long long init_key[],
                     unsigned long long key_length)
{
    unsigned long long i, j, k;
    init_genrand64(19650218ULL);
    i=1; j=0;
    k = (NN>key_length ? NN : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 62)) *
          3935559000370003845ULL))
          + init_key[j] + j; /* non linear */
        i++; j++;
        if (i>=NN) { mt[0] = mt[NN-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=NN-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 62)) *
          2862933555777941757ULL))
          - i; /* non linear */
        i++;
        if (i>=NN) { mt[0] = mt[NN-1]; i=1; }
    }

    mt[0] = 1ULL << 63; /* MSB is 1; assuring non-zero initial array */
}

/* generates a random number on [0, 2^64-1]-interval */
unsigned long long genrand64_int64(void)
{
    int i;
    unsigned long long x;
    static unsigned long long mag01[2]={0ULL, MATRIX_A};

    if (mti >= NN) { /* generate NN words at one time */

        /* if init_genrand64() has not been called, */
        /* a default initial seed is used     */
        if (mti == NN+1)
```

```
            init_genrand64(5489ULL);

        for (i=0;i<NN-MM;i++) {
            x = (mt[i]&UM)|(mt[i+1]&LM);
            mt[i] = mt[i+MM] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
        }
        for (;i<NN-1;i++) {
            x = (mt[i]&UM)|(mt[i+1]&LM);
            mt[i] = mt[i+(MM-NN)] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
        }
        x = (mt[NN-1]&UM)|(mt[0]&LM);
        mt[NN-1] = mt[MM-1] ^ (x>>1) ^ mag01[(int)(x&1ULL)];

        mti = 0;
    }

    x = mt[mti++];

    x ^= (x >> 29) & 0x5555555555555555ULL;
    x ^= (x << 17) & 0x71D67FFFEDA60000ULL;
    x ^= (x << 37) & 0xFFF7EEE000000000ULL;
    x ^= (x >> 43);

    return x;
}

/* generates a random number on [0, 2^63-1]-interval */
long long genrand64_int63(void)
{
    return (long long)(genrand64_int64() >> 1);
}

/* generates a random number on [0,1]-real-interval */
double genrand64_real1(void)
{
    return (genrand64_int64() >> 11) * (1.0/9007199254740991.0);
}

/* generates a random number on [0,1)-real-interval */
double genrand64_real2(void)
{
    return (genrand64_int64() >> 11) * (1.0/9007199254740992.0);
}

/* generates a random number on (0,1)-real-interval */
double genrand64_real3(void)
{
    return ((genrand64_int64() >> 12) + 0.5) *
            (1.0/4503599627370496.0);
}

int main (int argc, char *argv[]) {
    FILE *res_map, *res_cpu;
        int mapnum = 0;
        uint64_t i;
```

```
        uint64_t *AI, *TT;
        uint64_t AI3,AI2,AI1,AI0;

// Allocate array of TT values and array of AI values
    TT = (uint64_t *) malloc (1* sizeof (uint64_t));
    AI = (uint64_t *) malloc (1* sizeof (uint64_t));

    clock_t start = clock();
    AI0 = 0;
    AI1 = 0;
    AI2 = 0;
    AI3 = 0;

    //initialize Mersenne Twist
    init_genrand64(0xd0036009e7a8c44a); // Seed from random.org

    map_allocate (1);
    for(i=0;i<NUM;i++)
    {
        TT[0] = genrand64_int64();
        //  Call subroutine subr.mc on the MAP.
        subr (TT, AI, mapnum);
        switch(AI[0])
        {
             case 3:
                {
                AI3++;
                break;
                }
            case 2:
                {
                AI2++;
                break;
                }
            case 1:
                {
                AI1++;
                break;
                }
            case 0:
                {
                AI0++;
                break;
                }
        }


    }


    // Display runtime
    printf("Runtime: %f seconds OR %lld clocks\n",
        ((double)clock()-start)/CLOCKS_PER_SEC,clock()-start);

    // Print out the Algebraic Immunity of each Function
```

```
        printf("Listed below is the number of functions with each "
                "Algebraic Immunity\n\n");

        printf("AI = 3: %lld\n",AI3);
        printf("AI = 2: %lld\n",AI2);
        printf("AI = 1: %lld\n",AI1);
        printf("AI = 0: %lld\n",AI0);

    map_free (1);

    exit(0);

    }//int main (int argc, char *argv[]) {
```

subr.mc

```
//*****************************************************************
//
//  subr.mc  - MAP C subroutine to determine Algebraic Immunity
//
//       Author:         Eric McCay
//       Created:        July 25, 2011
//
//       Description:  This program calls Algebraic_Immunity.v, which
//                     determines the Algebraic Immunity of the
//                     function provided in Truth Table Form.
//
//*****************************************************************

#include <libmap.h>

void subr (uint64_t tt[], uint64_t ai[], int mapnum) {

// Declare two OBM banks in SRC-6, one to store 16 TT values and
//    the other to store the corresponding output AI values.
        OBM_BANK_A (TT, uint64_t, 1)
        OBM_BANK_B (AI, uint64_t, 1)
        uint64_t my64bit_in;
        uint64_t my64bit_out;

        // Get 1 TT value by DMAing FROM the CPU
        DMA_CPU (CM2OBM, TT, MAP_OBM_stripe(1,"A"), tt,
                 1, 1*sizeof(uint64_t), 0);
        wait_DMA (0);

        // Call the macro with the input TT, and store the
        // result in AI[0] to return to main.c
        my64bit_in = TT[0];
        my_operator (my64bit_in, &my64bit_out);
        AI[0] = my64bit_out;

// Return 1 AI value by DMAing TO the CPU
        DMA_CPU (OBM2CM, AI, MAP_OBM_stripe(1,"B"), ai,
```

```
                1, 1*sizeof(uint64_t), 0);
        wait_DMA (0);

}
```

3. Algebraic_Immunity.v

```
module Algebraic_Immunity(TT, AI, DONE, CLK, CLR, START);
//----------------------------------------------------------------------
// Algebraic_Immunity -  Verilog code to determine the algebraic
//                       immunity of the provided function.
//
// Created:       August 24, 2011
// Author:        Eric McCay and Jon T. Butler
//
// Inputs:        TT - Truth table of the function being tested
// Outputs:       AlgebraicImmunity - The algebraic immunity of the
//                                    tested function
//
//----------------------------------------------------------------------
//

parameter       n = 6;      // The number of variables.
localparam      N = 2**n;   // Max number of elements in TT.

input           CLK;
input           CLR;
input           START;
input   [63:0]  TT;         // The function under test
wire    [63:0]  TT;

reg     [63:0]  TT_reg;

output  [63:0]  AI;     // Algebraic Immunity of function under test
output          DONE;   // Indicates completion
reg DONE;
reg [63:0] AI;

integer i;

// Variables for simultaneous equation solving
// The SimultArray holds the simultaneous equations
// To solve.  It's structure for n=6 is too large to
// display in a readable manner.  It is an extension of
// the array for n=5, with the degree 1 term A6 included,
// and each degree 2 term that includes A6.

reg     [21:0]      SimultArray [0:63]; // array to hold
                                        // the 2^n equations
reg     [63:0]      A0Array;
reg     [63:0]      A1Array;   // of each bit of the truth table
reg     [63:0]      A2Array;   // - it is used to create the
reg     [63:0]      A3Array;   // SimultArray.
```

```
reg     [63:0]        A4Array;
reg     [63:0]        A5Array;
reg     [63:0]        A6Array;

reg     [n:0]         RowCounter; // Keeps track of which row is
                                  // being searched
reg     [n:0]         RowUpdate;  // Used to maintain the desired
                                  // position of the next row of interest
reg     [n:0]         ColCounter; // Tracks the column of interest

// Used for determining if the annihilator is of degree 1
reg     [21:0]        Row0Terms1;
reg     [21:0]        Row1Terms1;
reg     [21:0]        Row2Terms1;
reg     [21:0]        Row3Terms1;
reg     [21:0]        Row4Terms1;
reg     [21:0]        Row5Terms1;
reg     [21:0]        Row0Terms2;
reg     [21:0]        Row1Terms2;
reg     [21:0]        Row2Terms2;
reg     [21:0]        Row3Terms2;
reg     [21:0]        Row4Terms2;
reg     [21:0]        Row5Terms2;
reg     [21:0]        Row6Terms2;
reg     [21:0]        Row7Terms2;
reg     [21:0]        Row8Terms2;
reg     [21:0]        Row9Terms2;
reg     [21:0]        Row10Terms2;
reg     [21:0]        Row11Terms2;
reg     [21:0]        Row12Terms2;
reg     [21:0]        Row13Terms2;
reg     [21:0]        Row14Terms2;
reg     [21:0]        Row15Terms2;
reg     [21:0]        Row16Terms2;
reg     [21:0]        Row17Terms2;
reg     [21:0]        Row18Terms2;
reg     [21:0]        Row19Terms2;
reg     [21:0]        Row20Terms2;

reg                   CompTrack;  // Used to determine if the
                                  // complement has been checked

//state parameters
localparam Idle = 0;
localparam Init = 1;
localparam RowSearch1 = 2;
localparam RowFound1 = 3;
localparam RowSwap1 = 4;
localparam FindDegree1 = 5;
localparam UpdateDegree1 = 6;
localparam ComplementCheck = 7;
localparam RowSearch2 = 8;
localparam RowFound2 = 9;
localparam RowSwap2 = 10;
localparam FindDegree2 = 11;
```

```
localparam UpdateDegree2 = 12;
localparam Finish = 13;

reg [3:0] state;

always @(posedge CLK)
begin: statereg

    if (CLR)
        begin: Clearing

            state <= Idle;
            TT_reg <= TT;

            AI <= 0;

            DONE <= 1'b0;

            CompTrack <= 0;

            // Initialize the SimultArray to all 0's
            // And put the correct values in the
            // Arrays used to build SimultArray
            for(i = 0;i < N; i = i+1)
                begin
                    A0Array[i] <= 1;
                    A1Array[i] <= i%2;
                    A2Array[i] <= (i>>1)%2;
                    A3Array[i] <= (i>>2)%2;
                    A4Array[i] <= (i>>3)%2;
                    A5Array[i] <= (i>>4)%2;
                    A6Array[i] <= (i>>5)%2;
                    SimultArray[i] <= {22{1'b0}};
                end

        end // Clearing
    else
        begin
            case (state)
                // state code is shifted left to improve readability

                // Idle does nothing except start execution so that
                // the function and its complement can follow the
                // same path through the code
                Idle:
begin
    if(START)
        begin
            state <= Init;
        end
end //Idle

                //Init builds the SimultArray, which is then solved
                //to determine the lowest degree annihilator for the
                //function being tested.  It also initializes all
```

```
                //variables used to process the array.
                Init:
begin

    for(i = 0;i < N; i = i+1)
        begin
            SimultArray[i] <= {TT_reg[i]&A5Array[i]&A6Array[i],
                               TT_reg[i]&A4Array[i]&A6Array[i],
                               TT_reg[i]&A4Array[i]&A5Array[i],
                               TT_reg[i]&A3Array[i]&A6Array[i],
                               TT_reg[i]&A3Array[i]&A5Array[i],
                               TT_reg[i]&A3Array[i]&A4Array[i],
                               TT_reg[i]&A2Array[i]&A6Array[i],
                               TT_reg[i]&A2Array[i]&A5Array[i],
                               TT_reg[i]&A2Array[i]&A4Array[i],
                               TT_reg[i]&A2Array[i]&A3Array[i],
                               TT_reg[i]&A1Array[i]&A6Array[i],
                               TT_reg[i]&A1Array[i]&A5Array[i],
                               TT_reg[i]&A1Array[i]&A4Array[i],
                               TT_reg[i]&A1Array[i]&A3Array[i],
                               TT_reg[i]&A1Array[i]&A2Array[i],
                               TT_reg[i]&A6Array[i],
                               TT_reg[i]&A5Array[i],
                               TT_reg[i]&A4Array[i],
                               TT_reg[i]&A3Array[i],
                               TT_reg[i]&A2Array[i],
                               TT_reg[i]&A1Array[i],
                               TT_reg[i]&A0Array[i]};

        end

    RowCounter <= 0;
    RowUpdate <= 0;
    ColCounter <= 0;

    state <= RowSearch1;

end // Init

                // This state checks the rows 1 at a time, starting
                // after the last row to have been updated, and
                // attempts to find a 1 in the column of interest.
                RowSearch1:
begin
    // If this code executes, we have established
    // reduced row echelon form and are ready
    // to determine the lowest degree annihilator
    if(ColCounter == 7)
        begin
            Row0Terms1 <= SimultArray[0];
            Row1Terms1 <= SimultArray[1];
            Row2Terms1 <= SimultArray[2];
            Row3Terms1 <= SimultArray[3];
            Row4Terms1 <= SimultArray[4];
            Row5Terms1 <= SimultArray[5];
```

```
            state <= FindDegree1;
        end
    // If the next code executes (meaning we have counted
    // all rows) then AI is 1 because we have at least
    // one free variable, allowing us to produce
    // a degree 1 annihilator
    else if(RowCounter == N)
        begin
            AI <= 1;
            state <= Finish;
        end
    else if(((SimultArray[RowCounter]>>(ColCounter))%2) == 1)
        begin
            state <= RowFound1;
        end
    else
        begin
            RowCounter <= RowCounter + 1;
        end

end // RowSearch1

                // Entering this state means we found a row with a 1
                // in the column of interest.  We will use this row to
                // zero out the column of interest in all other rows.
                RowFound1:
begin

    // This for loop adds (xors) the found row with all other
    // rows that have a 1 in the column of interest to zero
    // them out.
    for(i = 0;i < N; i = i + 1)
        begin
            if((((SimultArray[i]>>(ColCounter))%2) == 1)
                &&(i != RowCounter))
                begin
                    SimultArray[i] <= (SimultArray[i] ^
                                       SimultArray[RowCounter]);
                end
        end

    //If true the row is in the right place
    if (RowCounter == RowUpdate)
        begin
            RowUpdate <= RowUpdate + 1;
            ColCounter <= ColCounter + 1;
            state <= RowSearch1;
            RowCounter <= RowUpdate + 1;
        end
    else // row is in the wrong place
        begin
            ColCounter <= ColCounter + 1;
            state <= RowSwap1;
        end
```

```
end // RowFound1

                // swaps the found row to the proper position
                RowSwap1:
begin

    SimultArray[RowCounter] <= SimultArray[RowUpdate];
    SimultArray[RowUpdate] <= SimultArray[RowCounter];

    RowUpdate <= RowUpdate + 1;
    RowCounter <= RowUpdate + 1;

    state <= RowSearch1;

end // RowSwap1

                // Counts the number of 1's in each row
                FindDegree1:
begin

    Row0Terms1 <= (Row0Terms1[6]+Row0Terms1[5]+Row0Terms1[4]
        +Row0Terms1[3]+Row0Terms1[2]+Row0Terms1[1]+Row0Terms1[0]);
    Row1Terms1 <= (Row1Terms1[6]+Row1Terms1[5]+Row1Terms1[4]
        +Row1Terms1[3]+Row1Terms1[2]+Row1Terms1[1]+Row1Terms1[0]);
    Row2Terms1 <= (Row2Terms1[6]+Row2Terms1[5]+Row2Terms1[4]
        +Row2Terms1[3]+Row2Terms1[2]+Row2Terms1[1]+Row2Terms1[0]);
    Row3Terms1 <= (Row3Terms1[6]+Row3Terms1[5]+Row3Terms1[4]
        +Row3Terms1[3]+Row3Terms1[2]+Row3Terms1[1]+Row3Terms1[0]);
    Row4Terms1 <= (Row4Terms1[6]+Row4Terms1[5]+Row4Terms1[4]
        +Row4Terms1[3]+Row4Terms1[2]+Row4Terms1[1]+Row4Terms1[0]);
    Row5Terms1 <= (Row5Terms1[6]+Row5Terms1[5]+Row5Terms1[4]
        +Row5Terms1[3]+Row5Terms1[2]+Row5Terms1[1]+Row5Terms1[0]);

    state <= UpdateDegree1;

end // FindDegree1

                // First, it checks to see if any row had two 1's. If
                // so, AI is 1 (those two variables combine to form
                // a degree one annihilator).  Otherwise, it goes to
                // RowSearch2, which is looking for a degree 2
                // annihilator
                UpdateDegree1:
begin
    if((Row0Terms1>1)||(Row1Terms1>1)||(Row2Terms1>1)||(Row3Terms1>1)
        ||(Row4Terms1>1)||(Row5Terms1>1))
        begin
            AI <= 1;
            state <= Finish;
        end
    else
        begin
            state <= RowSearch2;
        end
```

```
end //UpdateDegree1

                // This is similar to RowSearch1, except this is
                // looking for degree 2 annihilators
                RowSearch2:
begin
    // If this code executes, we have established
    // reduced row echelon form and are ready
    // to determine the lowest degree annihilator
    if(ColCounter == 22)
        begin
            Row0Terms2 <= SimultArray[0];
            Row1Terms2 <= SimultArray[1];
            Row2Terms2 <= SimultArray[2];
            Row3Terms2 <= SimultArray[3];
            Row4Terms2 <= SimultArray[4];
            Row5Terms2 <= SimultArray[5];
            Row6Terms2 <= SimultArray[6];
            Row7Terms2 <= SimultArray[7];
            Row8Terms2 <= SimultArray[8];
            Row9Terms2 <= SimultArray[9];
            Row10Terms2 <= SimultArray[10];
            Row11Terms2 <= SimultArray[11];
            Row12Terms2 <= SimultArray[12];
            Row13Terms2 <= SimultArray[13];
            Row14Terms2 <= SimultArray[14];
            Row15Terms2 <= SimultArray[15];
            Row16Terms2 <= SimultArray[16];
            Row17Terms2 <= SimultArray[17];
            Row18Terms2 <= SimultArray[18];
            Row19Terms2 <= SimultArray[19];
            Row20Terms2 <= SimultArray[20];
            state <= FindDegree2;
        end
    // If the next code executes (meaning we have counted
    // all rows) then AI is 1 because we have at least
    // one free variable, allowing us to produce
    // a degree 1 annihilator
    else if(RowCounter == N)
        begin
            AI <= 2;
            state <= ComplementCheck;
        end
    else if(((SimultArray[RowCounter]>>(ColCounter))%2) == 1)
        begin
            state <= RowFound2; // Found a 1
        end
    else // no 1 found, check the next row
        begin
            RowCounter <= RowCounter + 1;
        end

end // RowSearch2

                // Entering this state means we found a row with a 1
```

```
                // in the column of interest.  We will use this row to
                // zero out the column of interest in all other rows.
                RowFound2:
begin

    // This for loop adds (xors) the found row with all other
    // rows that have a 1 in the column of interest to zero
    // them out.
    for(i = 0;i < N; i = i + 1)
        begin
            if((((SimultArray[i]>>(ColCounter))%2) == 1)
                    &&(i != RowCounter))
                begin
                    SimultArray[i] <= (SimultArray[i] ^
                                       SimultArray[RowCounter]);
                end
        end

    //If true the row is in the right place
    if (RowCounter == RowUpdate)
        begin
            RowUpdate <= RowUpdate + 1;
            ColCounter <= ColCounter + 1;
            state <= RowSearch2;
            RowCounter <= RowUpdate + 1;
        end
    else
        begin
            ColCounter <= ColCounter + 1;
            state <= RowSwap2;
        end

end // RowFound2

                // Put the row in the proper place
                RowSwap2:
begin

    SimultArray[RowCounter] <= SimultArray[RowUpdate];
    SimultArray[RowUpdate] <= SimultArray[RowCounter];

    RowUpdate <= RowUpdate + 1;
    RowCounter <= RowUpdate + 1;

    state <= RowSearch2;

end // RowSwap2

                // Counts the number of 1's in each row
                FindDegree2:
begin

    Row0Terms2 <= (Row0Terms2[21]+Row0Terms2[20]+Row0Terms2[19]
        +Row0Terms2[18]+Row0Terms2[17]+Row0Terms2[16]+Row0Terms2[15]
        +Row0Terms2[14]+Row0Terms2[13]+Row0Terms2[12]+Row0Terms2[11]
```

```
    +Row0Terms2[10]+Row0Terms2[9]+Row0Terms2[8]+Row0Terms2[7]
    +Row0Terms2[6]+Row0Terms2[5]+Row0Terms2[4]+Row0Terms2[3]
    +Row0Terms2[2]+Row0Terms2[1]+Row0Terms2[0]);
Row1Terms2 <= (Row1Terms2[21]+Row1Terms2[20]+Row1Terms2[19]
    +Row1Terms2[18]+Row1Terms2[17]+Row1Terms2[16]+Row1Terms2[15]
    +Row1Terms2[14]+Row1Terms2[13]+Row1Terms2[12]+Row1Terms2[11]
    +Row1Terms2[10]+Row1Terms2[9]+Row1Terms2[8]+Row1Terms2[7]
    +Row1Terms2[6]+Row1Terms2[5]+Row1Terms2[4]+Row1Terms2[3]
    +Row1Terms2[2]+Row1Terms2[1]+Row1Terms2[0]);
Row2Terms2 <= (Row2Terms2[21]+Row2Terms2[20]+Row2Terms2[19]
    +Row2Terms2[18]+Row2Terms2[17]+Row2Terms2[16]+Row2Terms2[15]
    +Row2Terms2[14]+Row2Terms2[13]+Row2Terms2[12]+Row2Terms2[11]
    +Row2Terms2[10]+Row2Terms2[9]+Row2Terms2[8]+Row2Terms2[7]
    +Row2Terms2[6]+Row2Terms2[5]+Row2Terms2[4]+Row2Terms2[3]
    +Row2Terms2[2]+Row2Terms2[1]+Row2Terms2[0]);
Row3Terms2 <= (Row3Terms2[21]+Row3Terms2[20]+Row3Terms2[19]
    +Row3Terms2[18]+Row3Terms2[17]+Row3Terms2[16]+Row3Terms2[15]
    +Row3Terms2[14]+Row3Terms2[13]+Row3Terms2[12]+Row3Terms2[11]
    +Row3Terms2[10]+Row3Terms2[9]+Row3Terms2[8]+Row3Terms2[7]
    +Row3Terms2[6]+Row3Terms2[5]+Row3Terms2[4]+Row3Terms2[3]
    +Row3Terms2[2]+Row3Terms2[1]+Row3Terms2[0]);
Row4Terms2 <= (Row4Terms2[21]+Row4Terms2[20]+Row4Terms2[19]
    +Row4Terms2[18]+Row4Terms2[17]+Row4Terms2[16]+Row4Terms2[15]
    +Row4Terms2[14]+Row4Terms2[13]+Row4Terms2[12]+Row4Terms2[11]
    +Row4Terms2[10]+Row4Terms2[9]+Row4Terms2[8]+Row4Terms2[7]
    +Row4Terms2[6]+Row4Terms2[5]+Row4Terms2[4]+Row4Terms2[3]
    +Row4Terms2[2]+Row4Terms2[1]+Row4Terms2[0]);
Row5Terms2 <= (Row5Terms2[21]+Row5Terms2[20]+Row5Terms2[19]
    +Row5Terms2[18]+Row5Terms2[17]+Row5Terms2[16]+Row5Terms2[15]
    +Row5Terms2[14]+Row5Terms2[13]+Row5Terms2[12]+Row5Terms2[11]
    +Row5Terms2[10]+Row5Terms2[9]+Row5Terms2[8]+Row5Terms2[7]
    +Row5Terms2[6]+Row5Terms2[5]+Row5Terms2[4]+Row5Terms2[3]
    +Row5Terms2[2]+Row5Terms2[1]+Row5Terms2[0]);
Row6Terms2 <= (Row6Terms2[21]+Row6Terms2[20]+Row6Terms2[19]
    +Row6Terms2[18]+Row6Terms2[17]+Row6Terms2[16]+Row6Terms2[15]
    +Row6Terms2[14]+Row6Terms2[13]+Row6Terms2[12]+Row6Terms2[11]
    +Row6Terms2[10]+Row6Terms2[9]+Row6Terms2[8]+Row6Terms2[7]
    +Row6Terms2[6]+Row6Terms2[5]+Row6Terms2[4]+Row6Terms2[3]
    +Row6Terms2[2]+Row6Terms2[1]+Row6Terms2[0]);
Row7Terms2 <= (Row7Terms2[21]+Row7Terms2[20]+Row7Terms2[19]
    +Row7Terms2[18]+Row7Terms2[17]+Row7Terms2[16]+Row7Terms2[15]
    +Row7Terms2[14]+Row7Terms2[13]+Row7Terms2[12]+Row7Terms2[11]
    +Row7Terms2[10]+Row7Terms2[9]+Row7Terms2[8]+Row7Terms2[7]
    +Row7Terms2[6]+Row7Terms2[5]+Row7Terms2[4]+Row7Terms2[3]
    +Row7Terms2[2]+Row7Terms2[1]+Row7Terms2[0]);
Row8Terms2 <= (Row8Terms2[21]+Row8Terms2[20]+Row8Terms2[19]
    +Row8Terms2[18]+Row8Terms2[17]+Row8Terms2[16]+Row8Terms2[15]
    +Row8Terms2[14]+Row8Terms2[13]+Row8Terms2[12]+Row8Terms2[11]
    +Row8Terms2[10]+Row8Terms2[9]+Row8Terms2[8]+Row8Terms2[7]
    +Row8Terms2[6]+Row8Terms2[5]+Row8Terms2[4]+Row8Terms2[3]
    +Row8Terms2[2]+Row8Terms2[1]+Row8Terms2[0]);
Row9Terms2 <= (Row9Terms2[21]+Row9Terms2[20]+Row9Terms2[19]
    +Row9Terms2[18]+Row9Terms2[17]+Row9Terms2[16]+Row9Terms2[15]
    +Row9Terms2[14]+Row9Terms2[13]+Row9Terms2[12]+Row9Terms2[11]
    +Row9Terms2[10]+Row9Terms2[9]+Row9Terms2[8]+Row9Terms2[7]
```

```
    +Row9Terms2[6]+Row9Terms2[5]+Row9Terms2[4]+Row9Terms2[3]
    +Row9Terms2[2]+Row9Terms2[1]+Row9Terms2[0]);
Row10Terms2 <= (Row10Terms2[21]+Row10Terms2[20]+Row10Terms2[19]
    +Row10Terms2[18]+Row10Terms2[17]+Row10Terms2[16]
    +Row10Terms2[15]+Row10Terms2[14]+Row10Terms2[13]
    +Row10Terms2[12]+Row10Terms2[11]+Row10Terms2[10]
    +Row10Terms2[9]+Row10Terms2[8]+Row10Terms2[7]+Row10Terms2[6]
    +Row10Terms2[5]+Row10Terms2[4]+Row10Terms2[3]+Row10Terms2[2]
    +Row10Terms2[1]+Row10Terms2[0]);
Row11Terms2 <= (Row11Terms2[21]+Row11Terms2[20]+Row11Terms2[19]
    +Row11Terms2[18]+Row11Terms2[17]+Row11Terms2[16]
    +Row11Terms2[15]+Row11Terms2[14]+Row11Terms2[13]
    +Row11Terms2[12]+Row11Terms2[11]+Row11Terms2[10]
    +Row11Terms2[9]+Row11Terms2[8]+Row11Terms2[7]+Row11Terms2[6]
    +Row11Terms2[5]+Row11Terms2[4]+Row11Terms2[3]+Row11Terms2[2]
    +Row11Terms2[1]+Row11Terms2[0]);
Row12Terms2 <= (Row12Terms2[21]+Row12Terms2[20]+Row12Terms2[19]
    +Row12Terms2[18]+Row12Terms2[17]+Row12Terms2[16]
    +Row12Terms2[15]+Row12Terms2[14]+Row12Terms2[13]
    +Row12Terms2[12]+Row12Terms2[11]+Row12Terms2[10]
    +Row12Terms2[9]+Row12Terms2[8]+Row12Terms2[7]+Row12Terms2[6]
    +Row12Terms2[5]+Row12Terms2[4]+Row12Terms2[3]+Row12Terms2[2]
    +Row12Terms2[1]+Row12Terms2[0]);
Row13Terms2 <= (Row13Terms2[21]+Row13Terms2[20]+Row13Terms2[19]
    +Row13Terms2[18]+Row13Terms2[17]+Row13Terms2[16]
    +Row13Terms2[15]+Row13Terms2[14]+Row13Terms2[13]
    +Row13Terms2[12]+Row13Terms2[11]+Row13Terms2[10]
    +Row13Terms2[9]+Row13Terms2[8]+Row13Terms2[7]+Row13Terms2[6]
    +Row13Terms2[5]+Row13Terms2[4]+Row13Terms2[3]+Row13Terms2[2]
    +Row13Terms2[1]+Row13Terms2[0]);
Row14Terms2 <= (Row14Terms2[21]+Row14Terms2[20]+Row14Terms2[19]
    +Row14Terms2[18]+Row14Terms2[17]+Row14Terms2[16]
    +Row14Terms2[15]+Row14Terms2[14]+Row14Terms2[13]
    +Row14Terms2[12]+Row14Terms2[11]+Row14Terms2[10]
    +Row14Terms2[9]+Row14Terms2[8]+Row14Terms2[7]+Row14Terms2[6]
    +Row14Terms2[5]+Row14Terms2[4]+Row14Terms2[3]+Row14Terms2[2]
    +Row14Terms2[1]+Row14Terms2[0]);
Row15Terms2 <= (Row15Terms2[21]+Row15Terms2[20]+Row15Terms2[19]
    +Row15Terms2[18]+Row15Terms2[17]+Row15Terms2[16]
    +Row15Terms2[15]+Row15Terms2[14]+Row15Terms2[13]
    +Row15Terms2[12]+Row15Terms2[11]+Row15Terms2[10]
    +Row15Terms2[9]+Row15Terms2[8]+Row15Terms2[7]+Row15Terms2[6]
    +Row15Terms2[5]+Row15Terms2[4]+Row15Terms2[3]+Row15Terms2[2]
    +Row15Terms2[1]+Row15Terms2[0]);
Row16Terms2 <= (Row16Terms2[21]+Row16Terms2[20]+Row16Terms2[19]
    +Row16Terms2[18]+Row16Terms2[17]+Row16Terms2[16]
    +Row16Terms2[15]+Row16Terms2[14]+Row16Terms2[13]
    +Row16Terms2[12]+Row16Terms2[11]+Row16Terms2[10]
    +Row16Terms2[9]+Row16Terms2[8]+Row16Terms2[7]+Row16Terms2[6]
    +Row16Terms2[5]+Row16Terms2[4]+Row16Terms2[3]+Row16Terms2[2]
    +Row16Terms2[1]+Row16Terms2[0]);
Row17Terms2 <= (Row17Terms2[21]+Row17Terms2[20]+Row17Terms2[19]
    +Row17Terms2[18]+Row17Terms2[17]+Row17Terms2[16]
    +Row17Terms2[15]+Row17Terms2[14]+Row17Terms2[13]
    +Row17Terms2[12]+Row17Terms2[11]+Row17Terms2[10]
```

```
        +Row17Terms2[9]+Row17Terms2[8]+Row17Terms2[7]+Row17Terms2[6]
        +Row17Terms2[5]+Row17Terms2[4]+Row17Terms2[3]+Row17Terms2[2]
        +Row17Terms2[1]+Row17Terms2[0]);
    Row18Terms2 <= (Row18Terms2[21]+Row18Terms2[20]+Row18Terms2[19]
        +Row18Terms2[18]+Row18Terms2[17]+Row18Terms2[16]
        +Row18Terms2[15]+Row18Terms2[14]+Row18Terms2[13]
        +Row18Terms2[12]+Row18Terms2[11]+Row18Terms2[10]
        +Row18Terms2[9]+Row18Terms2[8]+Row18Terms2[7]+Row18Terms2[6]
        +Row18Terms2[5]+Row18Terms2[4]+Row18Terms2[3]+Row18Terms2[2]
        +Row18Terms2[1]+Row18Terms2[0]);
    Row19Terms2 <= (Row19Terms2[21]+Row19Terms2[20]+Row19Terms2[19]
        +Row19Terms2[18]+Row19Terms2[17]+Row19Terms2[16]
        +Row19Terms2[15]+Row19Terms2[14]+Row19Terms2[13]
        +Row19Terms2[12]+Row19Terms2[11]+Row19Terms2[10]
        +Row19Terms2[9]+Row19Terms2[8]+Row19Terms2[7]+Row19Terms2[6]
        +Row19Terms2[5]+Row19Terms2[4]+Row19Terms2[3]+Row19Terms2[2]
        +Row19Terms2[1]+Row19Terms2[0]);
    Row20Terms2 <= (Row20Terms2[21]+Row20Terms2[20]+Row20Terms2[19]
        +Row20Terms2[18]+Row20Terms2[17]+Row20Terms2[16]
        +Row20Terms2[15]+Row20Terms2[14]+Row20Terms2[13]
        +Row20Terms2[12]+Row20Terms2[11]+Row20Terms2[10]
        +Row20Terms2[9]+Row20Terms2[8]+Row20Terms2[7]+Row20Terms2[6]
        +Row20Terms2[5]+Row20Terms2[4]+Row20Terms2[3]+Row20Terms2[2]
        +Row20Terms2[1]+Row20Terms2[0]);

    state <= UpdateDegree2;

end // FindDegree2

                // First, it checks if there are two 1's in a row,
                // indicating a degree 2 annihilator.  If so, it goes
                // to ComplementCheck.  If not, it checks if the
                // complement has been tested.  If so and AI is still
                // set to 0, then AI is 3 (i.e. neither the function
                // nor its complement had a degree 2 or lower
                // annihilator.  Otherwise, it checks to see if this
                // is the complement and AI is currently set to 2.
                // If it is, we are done testing and AI is 2.  Default
                // is to go to ComplementCheck without adjusting AI,
                // which signals the the original function is being
                // tested and has no annihilator less then degree 3.
                UpdateDegree2:
begin
    if((Row0Terms2>1)||(Row1Terms2>1)||(Row2Terms2>1)||(Row3Terms2>1)
    ||(Row4Terms2>1)||(Row5Terms2>1)||(Row6Terms2>1)||(Row7Terms2>1)
    ||(Row8Terms2>1)||(Row9Terms2>1)||(Row10Terms2>1)||(Row11Terms2>1)
    ||(Row12Terms2>1)||(Row13Terms2>1)||(Row14Terms2>1)
    ||(Row15Terms2>1)||(Row16Terms2>1)||(Row17Terms2>1)
    ||(Row18Terms2>1)||(Row19Terms2>1)||(Row20Terms2>1))
        begin
            AI <= 2;
            state <= ComplementCheck;
        end
    else
        begin
```

```
            if((CompTrack == 1)&&(AI == 0))
                begin
                    AI <= 3;
                    state <= Finish;
                end
            else if((CompTrack == 1)&&(AI == 2))
                begin
                    state <= Finish;
                end
            else
                begin
                    state <= ComplementCheck;
                end
        end

end //UpdateDegree2

                // If AI is set to 2 and the complement has been
                // tested, we are done checking and can exit.
                // Otherwise, it complements the TT, sets the tracker
                // and starts back over at Init to test the complement.
                ComplementCheck:
begin
    if((AI == 2)&&(CompTrack == 1))
        begin
            state <= Finish;
        end
    else if(CompTrack == 0)
        begin
            TT_reg <= ~TT_reg;
            state <= Init;
            CompTrack <= 1;
        end
end //ComplementCheck

                // Testing complete.  Set DONE and exit.
                Finish:
begin
    DONE <= 1'b1;
end //Finish

            endcase

        end // state cases

end // statereg

endmodule
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. C SOURCE CODE

The source code to compute AI in C was developed utilizing the Verilog algorithm that first enumerated AI on the SRC-6. The code was compiled using Code::Blocks 10.05, and it was executed on a Windows 7 PC with 4 GB of RAM and an Intel® Core™2 Duo P8400 CPU operating at 2.26 GHz. The code is designed for single core operation and does not take advantage of the second core present in the processor.

All source code was formatted for presentation using Notepad++.

B.1 C SOURCE CODE (*n* = 4)

This source code is modeled after the simultaneous algorithm created for the SRC-6.

1. n4ai.c

```
//*********************************************************************
//
//  n4ai.c  -    C program to calculate Algebraic_Immunity (n=4)
//
//    Author:          Eric McCay
//    Created:         February 5, 2012
//
//    Description:  This program determines the Algebraic Immunity of
//                  all Boolean functions for a given n and provides an
//                  output specifying the number of functions with each
//                  AI.
//
//*********************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define NUM 65536  //number of values in TT 2^(2^n)

int main (int argc, char *argv[]) {

    clock_t start = clock(); // used for timing

    int i, j, k; // Some temporary variables

    int AI2;  // These 3 count functions with a particular AI
    int AI1;
    int AI0;

    AI0 = 2;
```

```
AI2 = 0;
AI1 = 0;

// Variables for simultaneous equation solving
// The SimultArray holds the simultaneous equations
// To solve.  It's structure for n=4 is:
//     A0 A1 A2 A3 A4
// g0  1  0  0  0  0 = 1
// g1  1  1  0  0  0 = 3
// g2  1  0  1  0  0 = 5
// g3  1  1  1  0  0 = 7
// g4  1  0  0  1  0 = 9
// g5  1  1  0  1  0 = 11
// g6  1  0  1  1  0 = 13
// g7  1  1  1  1  0 = 15
// g8  1  0  0  0  1 = 17
// g9  1  1  0  0  1 = 19
// g10 1  0  1  0  1 = 21
// g11 1  1  1  0  1 = 23
// g12 1  0  0  1  1 = 25
// g13 1  1  0  1  1 = 27
// g14 1  0  1  1  1 = 29
// g15 1  1  1  1  1 = 31

// The base Array is the default value that would go in an array
// if all values of the TT were 1.  The working array will receive
// a copy of this and then will have any lines where the function
// being tested has a 0 in the TT set to 0.
int AIBaseArray[] = {1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31};

int AIWorkingArray[]={1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31};

int WorkingAI = 0;
int ColCount = 0;
int RowUpdate = 0;
int RowCount = 0;

int Row0Terms = 0;
int Row1Terms = 0;
int Row2Terms = 0;
int Row3Terms = 0;

for(i=1;i<(NUM-1);i++) //We know that 1 and NUM-1 have AI=0
{
    // This portion of code is all to test the original function

    for(j=0;j<16;j++) // Fill in array with base
    {
        AIWorkingArray[j] = AIBaseArray[j];
    }

    WorkingAI = 0;
    RowUpdate = 0;
    RowCount = 0;
    ColCount = 0;
```

```
Row0Terms = 0;
Row1Terms = 0;
Row2Terms = 0;
Row3Terms = 0;

for(j=0;j<16;j++)
{
    if(((i>>j)%2)==0) //Zero out lines with 0 in TT
        {
            AIWorkingArray[j] = 0;
        }

}

while(ColCount<5)
{
    while(WorkingAI == 0) // we change the WorkingAI to exit
    {
        // Running this signifies that no empty columns have
        // been found.  We now determine if there are two 1's
        // in a row, signifying a degree 1 annihilator
        if(ColCount == 5)
        {
            // This counts the number of 1's in each row
            // Row 4 is ignored because there can be at most
            // 1 value in that row.
            for(j = 0; j < 5;j++)
            {
                Row0Terms += ((AIWorkingArray[0]>>j)%2);
                Row1Terms += ((AIWorkingArray[1]>>j)%2);
                Row2Terms += ((AIWorkingArray[2]>>j)%2);
                Row3Terms += ((AIWorkingArray[3]>>j)%2);
            }
            if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
                &&(Row3Terms<2))
            {
                WorkingAI = 2; // Found two 1's in a row
            }
            else
            {
                WorkingAI = 1; // Didn't find two 1's
            }
        }
        // free variable, so AI = 1
        else if(RowCount == 16)
        {
            WorkingAI = 1;
        }
        // 1 found in column of interest
        else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
        {
            for(j=0;j<16;j++) // zero out column
            {
                if(((AIWorkingArray[j]>>ColCount)%2==1)
```

```
                    &(j != RowCount)) //but not row of interest
                {
                    AIWorkingArray[j] = AIWorkingArray[j] ^
                                        AIWorkingArray[RowCount];
                }
            }
            if(RowCount != RowUpdate) // row in wrong position
            {
                k = AIWorkingArray[RowCount];
                AIWorkingArray[RowCount] =
                    AIWorkingArray[RowUpdate];
                AIWorkingArray[RowUpdate] = k;
            }
            RowUpdate++;
            RowCount = RowUpdate;
            ColCount++;
        }
        else
            {
                RowCount++;
            }
    }
    ColCount++;
}

if(WorkingAI == 2)
{
    WorkingAI = 0; // Used to make the complement test the same
}

//This portion of code is all to test the complement
if(WorkingAI != 1)
{
    for(j=0;j<16;j++) // Fill in array with base
    {
        AIWorkingArray[j] = AIBaseArray[j];
    }

    RowUpdate = 0;
    RowCount = 0;
    ColCount = 0;

    Row0Terms = 0;
    Row1Terms = 0;
    Row2Terms = 0;
    Row3Terms = 0;

    k = (NUM - 1)^i; //complement the input

    for(j=0;j<16;j++)
    {
        if(((k>>j)%2)==0) // Zero out lines with 0 in TT
            {
                AIWorkingArray[j] = 0;
            }
```

```
}

while(ColCount<5)
{
    while(WorkingAI == 0) // Done when AI changes
    {
        if(ColCount == 5) // Found no empty column
        {
            // Count the number of 1's in each row
            for(j = 0; j < 5;j++)
            {
                Row0Terms += ((AIWorkingArray[0]>>j)%2);
                Row1Terms += ((AIWorkingArray[1]>>j)%2);
                Row2Terms += ((AIWorkingArray[2]>>j)%2);
                Row3Terms += ((AIWorkingArray[3]>>j)%2);
            }
            if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
                &&(Row3Terms<2))
            {
                WorkingAI = 2; // No degree 1 annihilators
            }
            else
            {
                WorkingAI = 1; // two 1's in a row
            }
        }
        // Indicates a free variable, so AI = 1
        else if(RowCount == 16)
        {
            WorkingAI = 1;
        }
        // 1 found in column of interest
        else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
        {
            for(j=0;j<16;j++) // zero out column
            {
                if(((AIWorkingArray[j]>>ColCount)%2==1)
                &(j != RowCount)) //but not row of interest
                {
                    AIWorkingArray[j] = AIWorkingArray[j] ^
                                AIWorkingArray[RowCount];
                }
            }
            // Row is in the wrong position, so swap
            if(RowCount != RowUpdate)
            {
                k = AIWorkingArray[RowCount];
                AIWorkingArray[RowCount] =
                    AIWorkingArray[RowUpdate];
                AIWorkingArray[RowUpdate] = k;
            }
            RowUpdate++;
            RowCount = RowUpdate;
            ColCount++;
```

```
                    }
                    else
                        {
                            RowCount++;
                        }
                }
                ColCount++;
            }
        }

        if(WorkingAI == 2)
        {
            AI2++; // Means that no degree 1 annihilators found
        }
        else if(WorkingAI == 1)
        {
            AI1++;
        }

    }

    AI0 = 2; // There are 2 degree 0 annihilators for any number of
             // variables

    // Display runtime
    printf("Runtime: %f seconds OR %d clocks\n",
        ((double)clock()-start)/CLOCKS_PER_SEC,clock()-start);

// Print out the Algebraic Immunity of each Function
    printf("Listed below is the number of functions with each "
            "Algebraic Immunity\n\n");

        printf("AI = 2: %d\n",AI2);
        printf("AI = 1: %d\n",AI1);
        printf("AI = 0: %d\n",AI0);

    exit(0);

    }//int main (int argc, char *argv[]) {
```

B.2 C SOURCE CODE (n = 5)

This source code is an extension of the code for n = 4.

1. n5ai.c

```
//*********************************************************************
//
//  n5ai.c  -    C program to calculate Algebraic_Immunity (n=5)
//
//    Author:         Eric McCay
//    Created:        February 5, 2012
```

```
//
//    Description:  This program determines the Algebraic Immunity of
//                  all Boolean functions for a given n and provides an
//                  output specifying the number of functions with each
//                  AI.
//
//*********************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main (int argc, char *argv[]) {

    clock_t start = clock();

    long long NUM = 65536; //number of values in TT 2^(2^n)
    NUM = NUM*NUM;

    long long i, j, k; // Some temporary variables

    long long AI3;
    long long AI2;  // These 3 count functions with a particular AI
    long long AI1;
    long long AI0;

    AI0 = 2;
    AI1 = 0;
    AI2 = 0;
    AI3 = 0;

    // Variables for simultaneous equation solving
    // The SimultArray holds the simultaneous equations
    // To solve.  It's structure for n=5 is:
    //     A0 A1 A2 A3 A4 A5 A12 A13 A14 A15 A23 A24 A25 A34 A35 A45
    // g0   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g1   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g2   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g3   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g4   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g5   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g6   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g7   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g8   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g9   x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g10  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g11  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g12  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g13  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g14  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g15  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g16  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g17  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g18  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
    // g19  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
```

```
// g20  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g21  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g22  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g23  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g24  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g25  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g26  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g27  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g28  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g29  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g30  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x
// g31  x  x  x  x  x  x  x   x   x   x   x   x   x   x   x   x

// The base Array is the default value that would go in an array
// if all values of the TT were 1.  The working array will receive
// a copy of this and then will have any lines where the function
// being tested has a 0 in the TT set to 0.

int A0Array[32];
int A1Array[32];
int A2Array[32];
int A3Array[32];
int A4Array[32];
int A5Array[32];

int AIBaseArray[32];
int AIWorkingArray[32];

// Used to track AI for the function and its complement
int WorkingAI1 = 0;
int WorkingAI2 = 0;

// These track position in the matrix to put it in reduced
// row echelon form
int ColCount = 0;
int RowUpdate = 0;
int RowCount = 0;

// These are used to count the number of ones in a row once the
// matrix is in reduced row echelon form
int Row0Terms = 0;
int Row1Terms = 0;
int Row2Terms = 0;
int Row3Terms = 0;
int Row4Terms = 0;
int Row5Terms = 0;
int Row6Terms = 0;
int Row7Terms = 0;
int Row8Terms = 0;
int Row9Terms = 0;
int Row10Terms = 0;
int Row11Terms = 0;
int Row12Terms = 0;
int Row13Terms = 0;
int Row14Terms = 0;
```

```
// Populating the arrays that are used to create the base array
// and prints it as a binary
for(i=0;i<32;i++)
{
    A0Array[i]=1;
    A1Array[i]=i%2;
    A2Array[i]=(i>>1)%2;
    A3Array[i]=(i>>2)%2;
    A4Array[i]=(i>>3)%2;
    A5Array[i]=(i>>4)%2;
}

// This populates the base array
for(i=0;i<32;i++)
{
    AIBaseArray[i] = ((A4Array[i]&A5Array[i])<<15)+
                     ((A3Array[i]&A5Array[i])<<14)+
                     ((A3Array[i]&A4Array[i])<<13)+
                     ((A2Array[i]&A5Array[i])<<12)+
                     ((A2Array[i]&A4Array[i])<<11)+
                     ((A2Array[i]&A3Array[i])<<10)+
                     ((A1Array[i]&A5Array[i])<<9)+
                     ((A1Array[i]&A4Array[i])<<8)+
                     ((A1Array[i]&A3Array[i])<<7)+
                     ((A1Array[i]&A2Array[i])<<6)+
                     (A5Array[i]<<5)+
                     (A4Array[i]<<4)+
                     (A3Array[i]<<3)+
                     (A2Array[i]<<2)+
                     (A1Array[i]<<1)+
                     (A0Array[i]);
    printf("AIBaseArray[%d]: ",i);
    if((AIBaseArray[i]&0x8000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x4000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x2000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x1000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0800)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0400)>0)
        printf("1");
```

```
    else
        printf("0");
    if((AIBaseArray[i]&0x0200)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0100)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0080)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0040)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0020)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0010)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0008)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0004)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0002)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x0001)>0)
        printf("1\n");
    else
        printf("0\n");
}

for(i=1;i<(NUM-1);i++)
{
    if(((i-1)%25000000)==0)
        printf("Iteration %lld, AI3 = %lld, AI2 = %lld, "
        " AI1 = %lld, Runtime: %f seconds OR %lld clocks\n",
        i,AI3,AI2,AI1,
        ((double)clock()-start)/CLOCKS_PER_SEC,clock()-start);

    // This portion of code is all to test the original function
    // for degree 1 annihilators
```

```
// Fill in array with base
for(j=0;j<32;j++)
{
    AIWorkingArray[j] = AIBaseArray[j];
}

//Zero out all working variables
WorkingAI1 = 0;
WorkingAI2 = 0;
RowUpdate = 0;
RowCount = 0;
ColCount = 0;

Row0Terms = 0;
Row1Terms = 0;
Row2Terms = 0;
Row3Terms = 0;
Row4Terms = 0;

// Zero out lines with 0 in TT
for(j=0;j<32;j++)
{
    if(((i>>j)%2)==0)
        {
            AIWorkingArray[j] = 0;
        }

}

//Check for degree 1 annihilators
while(ColCount<6)
{
    //If we change AI this loop is done
    while(WorkingAI1 == 0)
    {
        // Signifies checking all degree 0 and 1 terms
        if(ColCount == 6)
        {
            // This adds up the bits in each row
            // if more than 1 number in a row,
            // there must be an annihilator
            for(j = 0; j < 6;j++)
            {
                Row0Terms += ((AIWorkingArray[0]>>j)%2);
                Row1Terms += ((AIWorkingArray[1]>>j)%2);
                Row2Terms += ((AIWorkingArray[2]>>j)%2);
                Row3Terms += ((AIWorkingArray[3]>>j)%2);
                Row4Terms += ((AIWorkingArray[4]>>j)%2);
            }
            if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
                &&(Row3Terms<2)&&(Row4Terms<2))
            {
                WorkingAI1 = 2;
            }
            else
```

```
                {
                    WorkingAI1 = 1;
                    WorkingAI2 = 1; // set both AIs to 1
                }
            }
            // Indicates a free variable exists
            else if(RowCount == 32)
            {
                WorkingAI1 = 1;
                WorkingAI2 = 1; // set both AIs to 1
            }
             // 1 found in column of interest
            else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
            {
                for(j=0;j<32;j++) // zero out column
                {
                    if(((AIWorkingArray[j]>>ColCount)%2==1)
                        &(j != RowCount)) //but not row of interest
                    {
                        AIWorkingArray[j] = AIWorkingArray[j] ^
                                            AIWorkingArray[RowCount];
                    }
                }
                // swap row if in wrong position
                if(RowCount != RowUpdate)
                {
                    k = AIWorkingArray[RowCount];
                    AIWorkingArray[RowCount] =
                        AIWorkingArray[RowUpdate];
                    AIWorkingArray[RowUpdate] = k;
                }
                // Move to next row and column
                RowUpdate++;
                RowCount = RowUpdate;
                ColCount++;
            }
            else
                {
                    RowCount++;
                }
        }
        ColCount++;
    }

    if(WorkingAI1 == 2)
    {
        WorkingAI1 = 0; // Used to make the degree 2 test the same
    }

    // This section tests original function for degree 2
    // annihilators
    if(WorkingAI1 != 1)
    {
        RowCount = RowUpdate;
        ColCount = 6; // Start at the correct column
```

```
Row0Terms = 0;
Row1Terms = 0;
Row2Terms = 0;
Row3Terms = 0;
Row4Terms = 0;
Row5Terms = 0;
Row6Terms = 0;
Row7Terms = 0;
Row8Terms = 0;
Row9Terms = 0;
Row10Terms = 0;
Row11Terms = 0;
Row12Terms = 0;
Row13Terms = 0;
Row14Terms = 0;

while(ColCount<16)
{
    while(WorkingAI1 == 0)
    {
        if(ColCount == 16)
        {
            // Found no empty columns, so count the number
            // of 1's in each row
            for(j = 0; j < 16;j++)
            {
                Row0Terms += ((AIWorkingArray[0]>>j)%2);
                Row1Terms += ((AIWorkingArray[1]>>j)%2);
                Row2Terms += ((AIWorkingArray[2]>>j)%2);
                Row3Terms += ((AIWorkingArray[3]>>j)%2);
                Row4Terms += ((AIWorkingArray[4]>>j)%2);
                Row5Terms += ((AIWorkingArray[5]>>j)%2);
                Row6Terms += ((AIWorkingArray[6]>>j)%2);
                Row7Terms += ((AIWorkingArray[7]>>j)%2);
                Row8Terms += ((AIWorkingArray[8]>>j)%2);
                Row9Terms += ((AIWorkingArray[9]>>j)%2);
                Row10Terms += ((AIWorkingArray[10]>>j)%2);
                Row11Terms += ((AIWorkingArray[11]>>j)%2);
                Row12Terms += ((AIWorkingArray[12]>>j)%2);
                Row13Terms += ((AIWorkingArray[13]>>j)%2);
                Row14Terms += ((AIWorkingArray[14]>>j)%2);
            }
            if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
            &&(Row3Terms<2)&&(Row4Terms<2)&&(Row5Terms<2)
            &&(Row6Terms<2)&&(Row7Terms<2)&&(Row8Terms<2)
            &&(Row9Terms<2)&&(Row10Terms<2)&&(Row11Terms<2)
            &&(Row12Terms<2)&&(Row13Terms<2)
            &&(Row14Terms<2))
            {
                WorkingAI1 = 3;
            }
            else
            {
                WorkingAI1 = 2;
```

```
                }
            }
            else if(RowCount == 32)
            {
                WorkingAI1 = 2;
            }
            // 1 found in column of interest
            else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
            {
                for(j=0;j<32;j++) // zero out column
                {
                    if(((AIWorkingArray[j]>>ColCount)%2==1)
                    &(j != RowCount)) //but not row of interest
                    {
                        AIWorkingArray[j] = AIWorkingArray[j] ^
                                    AIWorkingArray[RowCount];
                    }
                }
                // swap row if in wrong position
                if(RowCount != RowUpdate)
                {
                    k = AIWorkingArray[RowCount];
                    AIWorkingArray[RowCount] =
                        AIWorkingArray[RowUpdate];
                    AIWorkingArray[RowUpdate] = k;
                }
                RowUpdate++;
                RowCount = RowUpdate;
                ColCount++;
            }
            else
                {
                    RowCount++;
                }
        }
        ColCount++;
    }
}

//This portion of code is all to test the complement
//all sections operate the same as in the code above so there
//is less commenting
if(WorkingAI1 != 1)
{
    for(j=0;j<32;j++) // Fill in array with base
    {
        AIWorkingArray[j] = AIBaseArray[j];
    }

    RowUpdate = 0;
    RowCount = 0;
    ColCount = 0;

    Row0Terms = 0;
    Row1Terms = 0;
```

```
Row2Terms = 0;
Row3Terms = 0;
Row4Terms = 0;

k = (NUM - 1)^i; //complement the input

for(j=0;j<32;j++) // Zero out lines with 0 in TT
{
    if(((k>>j)%2)==0)
        {
            AIWorkingArray[j] = 0;
        }

}

while(ColCount<6)
{
    while(WorkingAI2 == 0)
    {
        if(ColCount == 6)
        {
            for(j = 0; j < 6;j++)
            {
                Row0Terms += ((AIWorkingArray[0]>>j)%2);
                Row1Terms += ((AIWorkingArray[1]>>j)%2);
                Row2Terms += ((AIWorkingArray[2]>>j)%2);
                Row3Terms += ((AIWorkingArray[3]>>j)%2);
                Row4Terms += ((AIWorkingArray[4]>>j)%2);
            }
            if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
                &&(Row3Terms<2)&&(Row4Terms<2))
            {
                WorkingAI2 = 2;
            }
            else
            {
                WorkingAI2 = 1;
            }
        }
        else if(RowCount == 32)
        {
            WorkingAI2 = 1;
        }
        // 1 found in column of interest
        else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
        {
            for(j=0;j<32;j++) // zero out column
            {
                if(((AIWorkingArray[j]>>ColCount)%2==1)
                &(j != RowCount)) //but not row of interest
                {
                    AIWorkingArray[j] = AIWorkingArray[j] ^
                                AIWorkingArray[RowCount];
                }
            }
```

```
                    // swap row if in wrong position
                    if(RowCount != RowUpdate)
                    {
                        k = AIWorkingArray[RowCount];
                        AIWorkingArray[RowCount] =
                            AIWorkingArray[RowUpdate];
                        AIWorkingArray[RowUpdate] = k;
                    }
                    RowUpdate++;
                    RowCount = RowUpdate;
                    ColCount++;
                }
                else
                    {
                        RowCount++;
                    }
            }
            ColCount++;
        }
    }

    if(WorkingAI2 == 2)
    {
        WorkingAI2 = 0; // Used to make the degree 2 test the same
    }

    // This section tests complement function for
    // degree 2 annihilators
    if((WorkingAI2 != 1)&&(WorkingAI1 != 1))
    {
        RowCount = RowUpdate;
        ColCount = 6;

        Row0Terms = 0;
        Row1Terms = 0;
        Row2Terms = 0;
        Row3Terms = 0;
        Row4Terms = 0;
        Row5Terms = 0;
        Row6Terms = 0;
        Row7Terms = 0;
        Row8Terms = 0;
        Row9Terms = 0;
        Row10Terms = 0;
        Row11Terms = 0;
        Row12Terms = 0;
        Row13Terms = 0;
        Row14Terms = 0;

        while(ColCount<16)
        {
            while(WorkingAI2 == 0)
            {
                if(ColCount == 16)
                {
```

```
        for(j = 0; j < 16;j++)
        {
            Row0Terms += ((AIWorkingArray[0]>>j)%2);
            Row1Terms += ((AIWorkingArray[1]>>j)%2);
            Row2Terms += ((AIWorkingArray[2]>>j)%2);
            Row3Terms += ((AIWorkingArray[3]>>j)%2);
            Row4Terms += ((AIWorkingArray[4]>>j)%2);
            Row5Terms += ((AIWorkingArray[5]>>j)%2);
            Row6Terms += ((AIWorkingArray[6]>>j)%2);
            Row7Terms += ((AIWorkingArray[7]>>j)%2);
            Row8Terms += ((AIWorkingArray[8]>>j)%2);
            Row9Terms += ((AIWorkingArray[9]>>j)%2);
            Row10Terms += ((AIWorkingArray[10]>>j)%2);
            Row11Terms += ((AIWorkingArray[11]>>j)%2);
            Row12Terms += ((AIWorkingArray[12]>>j)%2);
            Row13Terms += ((AIWorkingArray[13]>>j)%2);
            Row14Terms += ((AIWorkingArray[14]>>j)%2);
        }
        if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
        &&(Row3Terms<2)&&(Row4Terms<2)&&(Row5Terms<2)
        &&(Row6Terms<2)&&(Row7Terms<2)&&(Row8Terms<2)
        &&(Row9Terms<2)&&(Row10Terms<2)&&(Row11Terms<2)
        &&(Row12Terms<2)&&(Row13Terms<2)
        &&(Row14Terms<2))
        {
            WorkingAI2 = 3;
        }
        else
        {
            WorkingAI2 = 2;
        }
    }
    else if(RowCount == 32)
    {
        WorkingAI2 = 2;
    }
    // 1 found in column of interest
    else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
    {
        for(j=0;j<32;j++) // zero out column
        {
            if(((AIWorkingArray[j]>>ColCount)%2==1)
            &(j != RowCount)) //but not row of interest
            {
                AIWorkingArray[j] = AIWorkingArray[j] ^
                            AIWorkingArray[RowCount];
            }
        }
        // swap row if in wrong position
        if(RowCount != RowUpdate)
        {
            k = AIWorkingArray[RowCount];
            AIWorkingArray[RowCount] =
                AIWorkingArray[RowUpdate];
            AIWorkingArray[RowUpdate] = k;
```

```
                        }
                        RowUpdate++;
                        RowCount = RowUpdate;
                        ColCount++;
                    }
                    else
                        {
                            RowCount++;
                        }
                }
                ColCount++;
            }
        }


        if(WorkingAI2 < WorkingAI1)
        {
            WorkingAI1 = WorkingAI2;
        }

        if(WorkingAI1 == 3)
        {
            AI3++; // Means no degree 2 annihilators found
        }
        else if(WorkingAI1 == 2)
        {
            AI2++; // Means that no degree 1 annihilators found
        }
        else if(WorkingAI1 == 1)
        {
            AI1++;
        }

    }

    // It is known that there are exactly 2 degree 0 annihilators
    // for each number of variables
    AI0 = 2;

    // Display runtime
    printf("Runtime: %f seconds OR %lld clocks\n",
        ((double)clock()-start)/CLOCKS_PER_SEC,clock()-start);

// Print out the Algebraic Immunity of each Function
    printf("Listed below is the number of functions with each "
           "Algebraic Immunity\n\n");

    printf("AI = 3: %lld\n",AI3);
    printf("AI = 2: %lld\n",AI2);
    printf("AI = 1: %lld\n",AI1);
    printf("AI = 0: %lld\n",AI0);

    exit(0);

    }//int main (int argc, char *argv[]) {
```

B.3 C SOURCE CODE (n = 6)

This source code is an extension of the code for n = 5. It contains a version of the Mersenne Twister Pseudorandom number generator, which is used to perform the random trials for a Monte Carlo test.

1. n6ai.c

```
//*********************************************************************
//
//  n6ai.c  -    C program to calculate Algebraic_Immunity (n=6)
//
//    Author:          Eric McCay
//    Created:         February 5, 2012
//
//    Description:  This program determines the Algebraic Immunity of
//                  all Boolean functions for a given n and provides an
//                  output specifying the number of functions with each
//                  AI.
//
//*********************************************************************

/*
A C-program for MT19937-64 (2004/9/29 version).
Coded by Takuji Nishimura and Makoto Matsumoto.

This is a 64-bit version of Mersenne Twister pseudorandom number
generator.

Before using, initialize the state by using init_genrand64(seed)
or init_by_array64(init_key, key_length).

Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote
   products derived from this software without specific prior written
   permission.
```

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

   References:
   T. Nishimura, ``Tables of 64-bit Mersenne Twisters''
     ACM Transactions on Modeling and
     Computer Simulation 10. (2000) 348--357.
   M. Matsumoto and T. Nishimura,
     ``Mersenne Twister: a 623-dimensionally equidistributed
       uniform pseudorandom number generator''
     ACM Transactions on Modeling and
     Computer Simulation 8. (Jan. 1998) 3--30.

   Any feedback is very welcome.
   http://www.math.hiroshima-u.ac.jp/~m-mat/MT/emt.html
   email: m-mat @ math.sci.hiroshima-u.ac.jp (remove spaces)
*/

#include <stdio.h>
#include <time.h>

#define NN 312
#define MM 156
#define MATRIX_A 0xB5026F5AA96619E9ULL
#define UM 0xFFFFFFFF80000000ULL /* Most significant 33 bits */
#define LM 0x7FFFFFFFULL /* Least significant 31 bits */

/* The array for the state vector */
static unsigned long long mt[NN];
/* mti==NN+1 means mt[NN] is not initialized */
static int mti=NN+1;

/* initializes mt[NN] with a seed */
void init_genrand64(unsigned long long seed)
{
    mt[0] = seed;
    for (mti=1; mti<NN; mti++)
        mt[mti] =  (6364136223846793005ULL * (mt[mti-1] ^
                    (mt[mti-1] >> 62)) + mti);
}

/* initialize by an array with array-length */
/* init_key is the array for initializing keys */
/* key_length is its length */
void init_by_array64(unsigned long long init_key[],
             unsigned long long key_length)
```

```
{
    unsigned long long i, j, k;
    init_genrand64(19650218ULL);
    i=1; j=0;
    k = (NN>key_length ? NN : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 62)) *
          3935559000370003845ULL))
          + init_key[j] + j; /* non linear */
        i++; j++;
        if (i>=NN) { mt[0] = mt[NN-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=NN-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 62)) *
          2862933555777941757ULL))
          - i; /* non linear */
        i++;
        if (i>=NN) { mt[0] = mt[NN-1]; i=1; }
    }

    mt[0] = 1ULL << 63; /* MSB is 1; assuring non-zero initial array */
}

/* generates a random number on [0, 2^64-1]-interval */
unsigned long long genrand64_int64(void)
{
    int i;
    unsigned long long x;
    static unsigned long long mag01[2]={0ULL, MATRIX_A};

    if (mti >= NN) { /* generate NN words at one time */

        /* if init_genrand64() has not been called, */
        /* a default initial seed is used     */
        if (mti == NN+1)
            init_genrand64(5489ULL);

        for (i=0;i<NN-MM;i++) {
            x = (mt[i]&UM)|(mt[i+1]&LM);
            mt[i] = mt[i+MM] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
        }
        for (;i<NN-1;i++) {
            x = (mt[i]&UM)|(mt[i+1]&LM);
            mt[i] = mt[i+(MM-NN)] ^ (x>>1) ^ mag01[(int)(x&1ULL)];
        }
        x = (mt[NN-1]&UM)|(mt[0]&LM);
        mt[NN-1] = mt[MM-1] ^ (x>>1) ^ mag01[(int)(x&1ULL)];

        mti = 0;
    }

    x = mt[mti++];

    x ^= (x >> 29) & 0x5555555555555555ULL;
```

```
    x ^= (x << 17) & 0x71D67FFFEDA60000ULL;
    x ^= (x << 37) & 0xFFF7EEE000000000ULL;
    x ^= (x >> 43);

    return x;
}

/* generates a random number on [0, 2^63-1]-interval */
long long genrand64_int63(void)
{
    return (long long)(genrand64_int64() >> 1);
}

/* generates a random number on [0,1]-real-interval */
double genrand64_real1(void)
{
    return (genrand64_int64() >> 11) * (1.0/9007199254740991.0);
}

/* generates a random number on [0,1)-real-interval */
double genrand64_real2(void)
{
    return (genrand64_int64() >> 11) * (1.0/9007199254740992.0);
}

/* generates a random number on (0,1)-real-interval */
double genrand64_real3(void)
{
    return ((genrand64_int64() >> 12) + 0.5) *
            (1.0/4503599627370496.0);
}

int main (int argc, char *argv[]) {

    clock_t start = clock();

    long long NUM = 500000000; //number of iterations to perform

    long long i, j, k; // Some temporary variables

    long long TT;  // used as the random TT that will be tested

    long long AI3;
    long long AI2;  // These 3 count functions with a particular AI
    long long AI1;
    long long AI0;

    AI0 = 2;
    AI1 = 0;
    AI2 = 0;
    AI3 = 0;

    // Variables for simultaneous equation solving
    // The SimultArray holds the simultaneous equations
    // To solve.  It's structure for n=6 is too large to display
```

```
// nicely here.  It looks similar to that for n=5, but includes
// the degree 1 term A6 and all degree 2 terms that contain A6.

// The base Array is the default value that would go in an array
// if all values of the TT were 1.  The working array will receive
// a copy of this and then will have any lines where the function
// being tested has a 0 in the TT set to 0.

int A0Array[64];
int A1Array[64];
int A2Array[64];
int A3Array[64];
int A4Array[64];
int A5Array[64];
int A6Array[64];

int AIBaseArray[64];
int AIWorkingArray[64];

int WorkingAI1 = 0;
int WorkingAI2 = 0;
int ColCount = 0;
int RowUpdate = 0;
int RowCount = 0;

int Row0Terms = 0;
int Row1Terms = 0;
int Row2Terms = 0;
int Row3Terms = 0;
int Row4Terms = 0;
int Row5Terms = 0;
int Row6Terms = 0;
int Row7Terms = 0;
int Row8Terms = 0;
int Row9Terms = 0;
int Row10Terms = 0;
int Row11Terms = 0;
int Row12Terms = 0;
int Row13Terms = 0;
int Row14Terms = 0;
int Row15Terms = 0;
int Row16Terms = 0;
int Row17Terms = 0;
int Row18Terms = 0;
int Row19Terms = 0;
int Row20Terms = 0;

//initialize Mersenne Twist
init_genrand64(0xd0036009e7a8c44a); // Seed from random.org

// Initialize the arrays to create the base array
for(i=0;i<64;i++)
{
    A0Array[i]=1;
    A1Array[i]=i%2;
```

```
    A2Array[i]=(i>>1)%2;
    A3Array[i]=(i>>2)%2;
    A4Array[i]=(i>>3)%2;
    A5Array[i]=(i>>4)%2;
    A6Array[i]=(i>>5)%2;
}

// Creates the base array and prints it in binary form
for(i=0;i<64;i++)
{
    AIBaseArray[i] = ((A5Array[i]&A6Array[i])<<21)+
                     ((A4Array[i]&A6Array[i])<<20)+
                     ((A4Array[i]&A5Array[i])<<19)+
                     ((A3Array[i]&A6Array[i])<<18)+
                     ((A3Array[i]&A5Array[i])<<17)+
                     ((A3Array[i]&A4Array[i])<<16)+
                     ((A2Array[i]&A6Array[i])<<15)+
                     ((A2Array[i]&A5Array[i])<<14)+
                     ((A2Array[i]&A4Array[i])<<13)+
                     ((A2Array[i]&A3Array[i])<<12)+
                     ((A1Array[i]&A6Array[i])<<11)+
                     ((A1Array[i]&A5Array[i])<<10)+
                     ((A1Array[i]&A4Array[i])<<9)+
                     ((A1Array[i]&A3Array[i])<<8)+
                     ((A1Array[i]&A2Array[i])<<7)+
                     (A6Array[i]<<6)+
                     (A5Array[i]<<5)+
                     (A4Array[i]<<4)+
                     (A3Array[i]<<3)+
                     (A2Array[i]<<2)+
                     (A1Array[i]<<1)+
                     (A0Array[i]);
    printf("AIBaseArray[%d]: ",i);
    if((AIBaseArray[i]&0x200000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x100000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x80000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x40000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x20000)>0)
        printf("1");
    else
        printf("0");
    if((AIBaseArray[i]&0x10000)>0)
        printf("1");
```

```
else
    printf("0");
if((AIBaseArray[i]&0x8000)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x4000)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x2000)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x1000)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0800)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0400)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0200)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0100)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0080)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0040)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0020)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0010)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0008)>0)
    printf("1");
else
    printf("0");
if((AIBaseArray[i]&0x0004)>0)
```

```
            printf("1");
        else
            printf("0");
        if((AIBaseArray[i]&0x0002)>0)
            printf("1");
        else
            printf("0");
        if((AIBaseArray[i]&0x0001)>0)
            printf("1\n");
        else
            printf("0\n");
}

// Perform NUM random iterations
for(i=0;i<NUM;i++)
{
    if((i%25000000)==0)
        printf("Iteration %lld, AI3 = %lld, AI2 = %lld, "
        "AI1 = %lld, Runtime: %f seconds OR %lld clocks\n",
        i,AI3,AI2,AI1,
        ((double)clock()-start)/CLOCKS_PER_SEC,clock()-start);
    // This portion of code is all to test the original function
    // for degree 1 annihilators

    // Fill in array with base
    for(j=0;j<64;j++)
    {
        AIWorkingArray[j] = AIBaseArray[j];
    }

    //Zero out all working variables
    WorkingAI1 = 0;
    WorkingAI2 = 0;
    RowUpdate = 0;
    RowCount = 0;
    ColCount = 0;

    Row0Terms = 0;
    Row1Terms = 0;
    Row2Terms = 0;
    Row3Terms = 0;
    Row4Terms = 0;
    Row5Terms = 0;

    // Zero out lines with 0 in TT
    // Random code inputs here
    TT = genrand64_int64();
    for(j=0;j<64;j++)
    {
        if(((TT>>j)%2)==0)
            {
                AIWorkingArray[j] = 0;
            }

    }
```

```
//Check for degree 1 annihilators
while(ColCount<7)
{
    //If we change AI this loop is done
    while(WorkingAI1 == 0)
    {
        // Signifies checking all degree 0 and 1 terms
        if(ColCount == 7)
        {
            // This adds up the bits in each row
            // if more than 1 number in a row,
            // there must be an annihilator
            for(j = 0; j < 7;j++)
            {
                Row0Terms += ((AIWorkingArray[0]>>j)%2);
                Row1Terms += ((AIWorkingArray[1]>>j)%2);
                Row2Terms += ((AIWorkingArray[2]>>j)%2);
                Row3Terms += ((AIWorkingArray[3]>>j)%2);
                Row4Terms += ((AIWorkingArray[4]>>j)%2);
                Row5Terms += ((AIWorkingArray[5]>>j)%2);
            }
            if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
               &&(Row3Terms<2)&&(Row4Terms<2)&&(Row5Terms<2))
            {
                WorkingAI1 = 2; // No degree 1 annihilator
            }
            else
            {
                WorkingAI1 = 1;
                WorkingAI2 = 1; // set both AIs to 1
            }
        }
        // Empty column signifies free variable
        else if(RowCount == 64)
        {
            WorkingAI1 = 1;
            WorkingAI2 = 1; // to exit properly
        }
        // 1 found in column of interest
        else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
        {
            for(j=0;j<64;j++) // zero out column
            {
                if(((AIWorkingArray[j]>>ColCount)%2==1)
                &(j != RowCount)) //but not row of interest
                {
                    AIWorkingArray[j] = AIWorkingArray[j] ^
                                        AIWorkingArray[RowCount];
                }
            }
            // swap row if in wrong position
            if(RowCount != RowUpdate)
            {
                k = AIWorkingArray[RowCount];
```

```
                    AIWorkingArray[RowCount] =
                        AIWorkingArray[RowUpdate];
                    AIWorkingArray[RowUpdate] = k;
                }
                // Move to next row and column
                RowUpdate++;
                RowCount = RowUpdate;
                ColCount++;
            }
            else
                {
                    RowCount++;
                }
        }
        ColCount++;
    }

    if(WorkingAI1 == 2)
    {
        WorkingAI1 = 0; // Used to make the degree 2 test the same
    }

    // This section tests original function for
    // degree 2 annihilators
    if(WorkingAI1 != 1)
    {
        RowCount = RowUpdate;
        ColCount = 7;

        Row0Terms = 0;
        Row1Terms = 0;
        Row2Terms = 0;
        Row3Terms = 0;
        Row4Terms = 0;
        Row5Terms = 0;
        Row6Terms = 0;
        Row7Terms = 0;
        Row8Terms = 0;
        Row9Terms = 0;
        Row10Terms = 0;
        Row11Terms = 0;
        Row12Terms = 0;
        Row13Terms = 0;
        Row14Terms = 0;
        Row15Terms = 0;
        Row16Terms = 0;
        Row17Terms = 0;
        Row18Terms = 0;
        Row19Terms = 0;
        Row20Terms = 0;

        while(ColCount<22)
        {
            while(WorkingAI1 == 0) // exit when AI changes
            {
```

```
if(ColCount == 22)
{
    // No empty columns, so count then 1's in
    // each row
    for(j = 0; j < 22;j++)
    {
        Row0Terms += ((AIWorkingArray[0]>>j)%2);
        Row1Terms += ((AIWorkingArray[1]>>j)%2);
        Row2Terms += ((AIWorkingArray[2]>>j)%2);
        Row3Terms += ((AIWorkingArray[3]>>j)%2);
        Row4Terms += ((AIWorkingArray[4]>>j)%2);
        Row5Terms += ((AIWorkingArray[5]>>j)%2);
        Row6Terms += ((AIWorkingArray[6]>>j)%2);
        Row7Terms += ((AIWorkingArray[7]>>j)%2);
        Row8Terms += ((AIWorkingArray[8]>>j)%2);
        Row9Terms += ((AIWorkingArray[9]>>j)%2);
        Row10Terms += ((AIWorkingArray[10]>>j)%2);
        Row11Terms += ((AIWorkingArray[11]>>j)%2);
        Row12Terms += ((AIWorkingArray[12]>>j)%2);
        Row13Terms += ((AIWorkingArray[13]>>j)%2);
        Row14Terms += ((AIWorkingArray[14]>>j)%2);
        Row15Terms += ((AIWorkingArray[15]>>j)%2);
        Row16Terms += ((AIWorkingArray[16]>>j)%2);
        Row17Terms += ((AIWorkingArray[17]>>j)%2);
        Row18Terms += ((AIWorkingArray[18]>>j)%2);
        Row19Terms += ((AIWorkingArray[19]>>j)%2);
        Row20Terms += ((AIWorkingArray[20]>>j)%2);
    }
    if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
    &&(Row3Terms<2)&&(Row4Terms<2)&&(Row5Terms<2)
    &&(Row6Terms<2)&&(Row7Terms<2)&&(Row8Terms<2)
    &&(Row9Terms<2)&&(Row10Terms<2)&&(Row11Terms<2)
    &&(Row12Terms<2)&&(Row13Terms<2)
    &&(Row14Terms<2)&&(Row15Terms<2)
    &&(Row16Terms<2)&&(Row17Terms<2)
    &&(Row18Terms<2)&&(Row19Terms<2)
    &&(Row20Terms<2))
    { // no degree 2 annihilators found
        WorkingAI1 = 3;
    }
    else
    { // A degree 2 annihilator was found
        WorkingAI1 = 2;
    }
}
else if(RowCount == 64)
{
    WorkingAI1 = 2;
}
// 1 found in column of interest
else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
{
    for(j=0;j<64;j++) // zero out column
    {
        if(((AIWorkingArray[j]>>ColCount)%2==1)
```

```
                    &(j != RowCount)) //but not row of interest
                    {
                        AIWorkingArray[j] = AIWorkingArray[j] ^
                                    AIWorkingArray[RowCount];
                    }
                }
                // swap row if in wrong position
                if(RowCount != RowUpdate)
                {
                    k = AIWorkingArray[RowCount];
                    AIWorkingArray[RowCount] =
                        AIWorkingArray[RowUpdate];
                    AIWorkingArray[RowUpdate] = k;
                }
                RowUpdate++;
                RowCount = RowUpdate;
                ColCount++;
            }
            else
                {
                    RowCount++;
                }
        }
        ColCount++;
    }
}

//This portion of code is all to test the complement
//It functions the same as the previous code so it has
//less commenting
if(WorkingAI1 != 1)
{
    for(j=0;j<64;j++) // Fill in array with base
    {
        AIWorkingArray[j] = AIBaseArray[j];
    }

    RowUpdate = 0;
    RowCount = 0;
    ColCount = 0;

    Row0Terms = 0;
    Row1Terms = 0;
    Row2Terms = 0;
    Row3Terms = 0;
    Row4Terms = 0;
    Row5Terms = 0;

    k = ~TT; //complement the input

    for(j=0;j<64;j++)
    {
        if(((k>>j)%2)==0) // Zero out lines with 0 in TT
            {
                AIWorkingArray[j] = 0;
```

```
        }

}

while(ColCount<7)
{
    while(WorkingAI2 == 0)
    {
        if(ColCount == 7)
        {
            for(j = 0; j < 7;j++)
            {
                Row0Terms += ((AIWorkingArray[0]>>j)%2);
                Row1Terms += ((AIWorkingArray[1]>>j)%2);
                Row2Terms += ((AIWorkingArray[2]>>j)%2);
                Row3Terms += ((AIWorkingArray[3]>>j)%2);
                Row4Terms += ((AIWorkingArray[4]>>j)%2);
                Row5Terms += ((AIWorkingArray[5]>>j)%2);
            }
            if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
            &&(Row3Terms<2)&&(Row4Terms<2)&&(Row5Terms<2))
            {
                WorkingAI2 = 2;
            }
            else
            {
                WorkingAI2 = 1;
            }
        }
        else if(RowCount == 64)
        {
            WorkingAI2 = 1;
        }
        // 1 found in column of interest
        else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
        {
            for(j=0;j<64;j++) // zero out column
            {
                if(((AIWorkingArray[j]>>ColCount)%2==1)
                &(j != RowCount)) //but not row of interest
                {
                    AIWorkingArray[j] = AIWorkingArray[j] ^
                                AIWorkingArray[RowCount];
                }
            }
            // swap row if in wrong position
            if(RowCount != RowUpdate)
            {
                k = AIWorkingArray[RowCount];
                AIWorkingArray[RowCount] =
                    AIWorkingArray[RowUpdate];
                AIWorkingArray[RowUpdate] = k;
            }
            RowUpdate++;
            RowCount = RowUpdate;
```

```
                ColCount++;
            }
            else
                {
                    RowCount++;
                }
        }
        ColCount++;
    }
}

if(WorkingAI2 == 2)
{
    WorkingAI2 = 0; // Used to make the degree 2 test the same
}

// This section tests complement function for
// degree 2 annihilators
if((WorkingAI2 != 1)&&(WorkingAI1 != 1))
{
    RowCount = RowUpdate;
    ColCount = 7;

    Row0Terms = 0;
    Row1Terms = 0;
    Row2Terms = 0;
    Row3Terms = 0;
    Row4Terms = 0;
    Row5Terms = 0;
    Row6Terms = 0;
    Row7Terms = 0;
    Row8Terms = 0;
    Row9Terms = 0;
    Row10Terms = 0;
    Row11Terms = 0;
    Row12Terms = 0;
    Row13Terms = 0;
    Row14Terms = 0;
    Row15Terms = 0;
    Row16Terms = 0;
    Row17Terms = 0;
    Row18Terms = 0;
    Row19Terms = 0;
    Row20Terms = 0;

    while(ColCount<22)
    {
        while(WorkingAI2 == 0)
        {
            if(ColCount == 22)
            {
                for(j = 0; j < 22;j++)
                {
                    Row0Terms += ((AIWorkingArray[0]>>j)%2);
                    Row1Terms += ((AIWorkingArray[1]>>j)%2);
```

```
            Row2Terms += ((AIWorkingArray[2]>>j)%2);
            Row3Terms += ((AIWorkingArray[3]>>j)%2);
            Row4Terms += ((AIWorkingArray[4]>>j)%2);
            Row5Terms += ((AIWorkingArray[5]>>j)%2);
            Row6Terms += ((AIWorkingArray[6]>>j)%2);
            Row7Terms += ((AIWorkingArray[7]>>j)%2);
            Row8Terms += ((AIWorkingArray[8]>>j)%2);
            Row9Terms += ((AIWorkingArray[9]>>j)%2);
            Row10Terms += ((AIWorkingArray[10]>>j)%2);
            Row11Terms += ((AIWorkingArray[11]>>j)%2);
            Row12Terms += ((AIWorkingArray[12]>>j)%2);
            Row13Terms += ((AIWorkingArray[13]>>j)%2);
            Row14Terms += ((AIWorkingArray[14]>>j)%2);
            Row15Terms += ((AIWorkingArray[15]>>j)%2);
            Row16Terms += ((AIWorkingArray[16]>>j)%2);
            Row17Terms += ((AIWorkingArray[17]>>j)%2);
            Row18Terms += ((AIWorkingArray[18]>>j)%2);
            Row19Terms += ((AIWorkingArray[19]>>j)%2);
            Row20Terms += ((AIWorkingArray[20]>>j)%2);
        }
        if((Row0Terms<2)&&(Row1Terms<2)&&(Row2Terms<2)
        &&(Row3Terms<2)&&(Row4Terms<2)&&(Row5Terms<2)
        &&(Row6Terms<2)&&(Row7Terms<2)&&(Row8Terms<2)
        &&(Row9Terms<2)&&(Row10Terms<2)&&(Row11Terms<2)
        &&(Row12Terms<2)&&(Row13Terms<2)
        &&(Row14Terms<2)&&(Row15Terms<2)
        &&(Row16Terms<2)&&(Row17Terms<2)
        &&(Row18Terms<2)&&(Row19Terms<2)
        &&(Row20Terms<2))
        {
            WorkingAI2 = 3;
        }
        else
        {
            WorkingAI2 = 2;
        }
    }
    else if(RowCount == 64)
    {
        WorkingAI2 = 2;
    }
    // 1 found in column of interest
    else if((AIWorkingArray[RowCount]>>ColCount)%2==1)
    {
        for(j=0;j<64;j++) // zero out column
        {
            if(((AIWorkingArray[j]>>ColCount)%2==1)
            &(j != RowCount)) //but not row of interest
            {
                AIWorkingArray[j] = AIWorkingArray[j] ^
                            AIWorkingArray[RowCount];
            }
        }
        // swap row if in wrong position
        if(RowCount != RowUpdate)
```

```
                        {
                            k = AIWorkingArray[RowCount];
                            AIWorkingArray[RowCount] =
                                AIWorkingArray[RowUpdate];
                            AIWorkingArray[RowUpdate] = k;
                        }
                        RowUpdate++;
                        RowCount = RowUpdate;
                        ColCount++;
                    }
                    else
                        {
                            RowCount++;
                        }
                }
                ColCount++;
            }
        }


        if(WorkingAI2 < WorkingAI1)
        {
            WorkingAI1 = WorkingAI2;
        }

        if(WorkingAI1 == 3)
        {
            AI3++; // Means no degree 2 annihilators found
        }
        else if(WorkingAI1 == 2)
        {
            AI2++; // Means that no degree 1 annihilators found
        }
        else if(WorkingAI1 == 1)
        {
            AI1++;
        }

    }

    // It is known that there are two functions with AI = 0 for any
    // number of variables
    AI0 = 2;

    // Display the runtime
    printf("Runtime: %f seconds OR %lld clocks\n",
        ((double)clock()-start)/CLOCKS_PER_SEC,clock()-start);

/*  Print out the Algebraic Immunity of each Function   */
    printf("Listed below is the number of functions with each "
            "Algebraic Immunity\n\n");

    printf("AI = 3: %lld\n",AI3);
    printf("AI = 2: %lld\n",AI2);
    printf("AI = 1: %lld\n",AI1);
```

```
printf("AI = 0: %lld\n",AI0);

exit(0);

}//int main (int argc, char *argv[]) {
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] N. Courtois, "Higher order correlation attacks, XL algorithm and cryptanalysis of toyocrypt RID C-6223-2009," *Information Security and Cryptology* - vol. 2587, pp. 182-199, 2002. Available: http://eprint.iacr.org/2002/087.pdf.

[2] N. Courtois and W. Meier, "Algebraic attacks on stream ciphers with linear feedback RID C-6223-2009," *Advances in Cryptology-Eurocrypt 2003*, vol. 2656, pp. 345-359, 2003. Available: http://www.nicolascourtois.me.uk/toyolili.pdf.

[3] J. Erickson, J. Ding and C. Christensen, "Algebraic Cryptanalysis of SMS4: Grobner Basis Attack and SAT Attack Compared," *Information Security and Cryptology*, vol. 5984, pp. 73-86, 2009. Available: http://www.nku.edu/~christensen/SMS4%20jeremy.pdf.

[4] J. Faugere, A. Otmani, L. Perret and J. Tillich, "Algebraic Cryptanalysis of McEliece Variants with Compact Keys," *Advances in Cryptology*, vol. 6110, pp. 279-298, 2010.

[5] Anonymous "The application usage and risk report," Palo Alto Networks, 2011. Available: http://www.paloaltonetworks.com/researchcenter/reports/.

[6] C. J. Etherington, "An analysis of cryptographically significant Boolean functions with high correlation immunity by reconfigurable computer," M.S. thesis, ECE Dept., NPS, Monterey, CA, 2010. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a536393.pdf.

[7] C. D. Johnson, "The circular pipeline: achieving higher throughput in the search for bent functions," M.S. thesis, ECE Dept., NPS, Monterey, CA, 2010. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a531571.pdf.

[8] T. R. O'Dowd, "Discovery of bent functions using the Fast Walsh Transform," M.S. thesis, ECE Dept., NPS, Monterey, CA, 2010. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a536595.pdf.

[9] J. L. Shafer, "An analysis of bent function properties using the transeunt triangle and the SRC-6 reconfigurable computer," M.S. thesis, ECE Dept., NPS, Monterey, CA, 2009. Available: http://www.dtic.mil/dtic/tr/fulltext/u2/a510033.pdf.

[10] J. Oetting and K. King, "The impact of IPSEC on DoD teleport throughput efficiency," *IEEE Military Communications Conference*, vol. 2, pp. 717-721, 2004.

[11] K. Rohwer and T. Krout, *Multiple Levels of Security in Support of Highly Mobile Tactical Internets - ELB ACTD.* 2001.

[12] A. Papantonopoulou, *Algebra: Pure & Applied.* Upper Saddle River, NJ: Prentice Hall, 2002.

[13] W. Meier, E. Pasalic and C. Carlet, "Algebraic attacks and decomposition of boolean functions," *Proc. Advances in Cryptology - Eurocrypt*, vol. 3027, pp. 474-491, 2004. Available: http://www.iacr.org/cryptodb/archive/2004/EUROCRYPT/2645/2645.pdf.

[14] J. L. Shafer, S. W. Schneider, J. T. Butler and P. Stanica, "Enumeration of bent boolean functions by reconfigurable computer," in *the 18th Annual International IEEE Symposium on Field-Programmable Custom Computing Machines*, Charlotte, NC, 2010, pp. 265-272, Available: http://faculty.nps.edu/butler/PDF/2010/Schafer_et_al_Bent.pdf.

[15] F. Armknecht, C. Carlet, P. Gaborit, S. Kuenzli, W. Meier and O. Ruatta, "Efficient computation of algebraic immunity for algebraic and fast algebraic attacks," *Proc. Advances in Cryptology - Eurocrypt*, vol. 4004, pp. 147-164, 2006. Available: http://www.unilim.fr/pages_perso/philippe.gaborit/AI_main.pdf.

[16] M. Albrecht, "Algebraic attacks on the Courtois toy cipher," *Cryptologia*, vol. 32, pp. 220-276, 2008. Available: http://www.sagemath.org/files/thesis/albrecht-thesis-2006.pdf.

[17] Y. D. Ziran Tu, "Algebraic immunity hierarchy of boolean functions," Cryptology Eprint Archive, Tech. Rep. 2007/259, 2007. Available: http://eprint.iacr.org/2007/259.pdf.

[18] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3-30, Jan, 1998. Available: http://doi.acm.org/10.1145/272991.272995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Clark Robertson
   Naval Postgraduate School
   Monterey, California

4. Dr. Jon T. Butler
   Naval Postgraduate School
   Monterey, California

5. Dr. Pantelimon Stanica
   Naval Postgraduate School
   Monterey, California

6. LT Eric McCay
   Naval Postgraduate School
   Monterey, California

7. Kimberly McCay
   Monterey, California

8. Judy McCay
   Arkansas State University
   Lake City, Arkansas

9. Thomas McCay
   Lake City, Arkansas

10. Sarah McCay
    University of Tennessee
    Memphis, Tennessee

11. Denise Brumpton
    Edmonds, Washington

12. Dr. John G. Harkins
National Security Agency
Fort Meade, Maryland

13. Dr. David R. Podany
National Security Agency
Fort Meade, Maryland

14. Mr. David Caliga
SRC Computers
Colorado Springs, Colorado

15. Mr. Jon Huppenthal
SRC Computers
Colorado Springs, Colorado

16. Dr. Jeff Hammes
SRC Computers
Colorado Springs, Colorado

17. LT Stuart Schneider
United States Navy
Sasebo, Japan